Peter P. Hofmann, Andy Schürr (Eds.)

OMER –
Object-oriented Modeling of
Embedded Real-Time Systems

Postworkshop Proceedings of

OMER-1: May 28/29 1999, Herrsching am Ammersee
OMER-2: May 10-12 2001, Herrsching am Ammersee

Gesellschaft für Informatik 2002

Lecture Notes in Informatics (LNI) - Proceedings
Series of the German Informatics society (GI)

Volume P-5

ISBN 3-88579-337-7
ISSN 1617-5468

Volume Editors
Dr. Peter Hofmann
DaimlerChrysler AG
HPC: T724, Werk 96
D-70546 Stuttgart, Germany

Prof. Dr. Andy Schürr
Technische Universität Darmstadt
Realtime Systems Lab (FG Echtzeitsysteme), FB 18
Merckstr. 25, D-64283 Darmstadt, Germany
Email: schuerr@rs.tu-darmstadt.de

Series Editorial Board
Heinrich C. Mayr, Universität Klagenfurt, Austria (Chairman, mayr@ifit.uni-klu.ac.at)
Jörg Becker, Universität Münster, Germany
Ulrich Furbach, Universität Koblenz, Germany
Axel Lehmann, Universität der Bundeswehr München, Germany
Peter Liggesmeyer, Universität Potsdam, Germany
Ernst W. Mayr, Technische Universität München, Germany
Heinrich Müller, Universität Dortmund, Germany
Heinrich Reinermann, Hochschule für Verwaltungswissenschaften Speyer, Germany
Karl-Heinz Rödiger, Universität Bremen, Germany
Sigrid Schubert, Universität Dortmund, Germany

Dissertations
Dorothea Wagner, Universität Konstanz, Germany

Seminars
Reinhard Wilhelm, Universität des Saarlandes, Germany

© Gesellschaft für Informatik, Bonn 2002
printed by Köllen Druck+Verlag GmbH, Bonn

Foreword

In today's world embedded realtime systems are everywhere. Often more or less unnoticed they fulfill the task to control the behavior of technical systems in our homes, cars, and offices, in the hospital and at various other places. In all these cases the complexity and number of functions realized in software increases rapidly. Often this software is still developed using software engineering technologies of the eighties. As a consequence, it does not fulfill our expectations concerning the needed quality expressed in terms of reliability, security, timeliness, maintainability, reusability, and other "ilities". Furthermore, it often suffers from a separation of functions and data as well as from the lack of well-defined subsystem boundaries with precisely documented interfaces.

These are the reasons why object-oriented and component-based methods are nowadays more or less accepted means for the development of embedded RT system software. They promise to facilitate the development, deployment, and reuse of software components with well-defined interfaces. This line of development started at the end of the last millenium with the announcement of the first generation of CASE tools that were able to generate code for embedded system targets from high-level models e.g. defined in the Unified Modeling Language (UML).

At that time we organized the first OMER workshop on "Object-oriented Modelling of Embedded Realtime systems" (*http://ist.unibw-muenchen.de/GROOM/OMER*). This workshop addressed all aspects of the development and application of object-oriented methods (languages, tools, and processes) for the analysis, design, and implementation of embedded RT software. It served as a platform for academia and industry as well as for tool developers and their clients to exchange their first experiences with OO development of embedded RT systems and to discuss new trends concerning e.g. the development of an UML standard extension (profile) for realtime modeling purposes.

First, OMER was planned to be a small workshop of the German Society of Informatics (GI) with about 20 to 30 participants. But after a while we had to recognize that it was not feasible to keep the number of participants as small as intended - finally about 60 persons from six different countries attended the workshop. Quite a number of researchers from different countries were very interested to present their products, experiences, and future plans in this area. Therefore, we had to switch the workshop language from German to English and to rearrange many other things. Finally, the workshop proceedings contained a total of 28 short position papers. Based on these contributions we were able to organize sessions addressing topics such as "RT System Architectures", "Unified Modeling Language Extensions", "RT CASE Tools and Programming Languages". Furthermore, we were especially honored that two distinguished experts accepted the invitation to present their latest research activities in this area. David Harel (among other things Dean of the Faculty of Mathematics and Computer Science at the Weizmann Institute of Science and cofounder of I-Logix Inc.) gave a talk "On the Behavioral Modeling of Complex Object-Oriented Systems", whereas Bran Selic (at that time Vice President of Advanced Technology at Objectime Limited) spoke about "Using UML to Model Complex Real-Time Architectures".

Two years later on we organized a second OMER workshop (*http://ist.unibw-muenchen.de/-GROOM/OMER-2*) at the same place - the conference building of the Bavarian Farmers Association in Herrsching am Ammersee nearby Munich. This time about 80 participants from all over the world used the workshop to discuss new experiences, insights, plans, etc. concerning the object-oriented development of embedded RT systems. It is worth-while to notice that - in contrast to the predecessor workshop - the accepted submissions builded clusters with respect to the addressed application domains: (1) automotive systems, (2) automation and production control systems, and (3) telecommunication systems. Furthermore, the topic of component-oriented modeling languages played a more important role than two years ago; about one half of the accepted 16 presentations dealt with component-oriented technologies for software development purposes.

In addition to the regular talks related to accepted position papers there were also a number of invited talks of distinguished RT system development experts: Bruce Douglas (Chief Evangelist of I-Logix Inc.), Morgan Björkander (method specialist from Telelogic AB), and Michael Kircher (senior engineer at Siemens AG Corporate Technology). They addressed the following topics: ”Using the UML Profile for Schedulability, Performance and Time”, ”Real-time Systems and the UML”, and ”Using Real-time CORBA effectively”.

Furthermore, we organized a number of tutorials, where representatives from Rational Software Corp., Telelogic AB, and ARTiSAN Software presented the latest features of their CASE tools, as well as two panel sessions. The first one was about ”OO Development of Distributed Systems, a Critical Assessment” with Manfred Broy (TU München) as Chair and with Theodor Tempelmeier (FH Rosenheim), Martin Wirsing (LMU München), and Jürgen Ziegler (Nokia Helsinki) on the panelists. The second late evening panel about ”The Missing Concepts of UML for Designing Embedded RT Systems” was organized as a kind of talk show. Peter Hruschka (System-Bauhaus) and Chris Rupp (SOPHIST GmbH) acted as moderators, whereas Morgan Björkander (Telelogic AB), Georg Färber (TU München), and Volker Kopetzky (Rational Software Corp.) were the to be interviewed guests.

One highlight of the OMER-2 workshop was the ”Object-Oriented Realtime Modeling Contest” organized and sponsored by DaimlerChrysler Research & Technology. The challenge of this contest was to design a typical automotive control function using an object-oriented technique. Starting with about 80 registered participating parties the contest was a great success. Finally, three different groups from the University of Northampton (place 3), Validas Model Validation AG (place 2) and St. Petersburg State Technical University/XJ Technologies (winners of the contest) were invited to present their solutions at the workshop.

Right after the second workshop we decided to ask all speakers of both OMER happenings to submit longer updated versions of their research activities presented at the workshops. After another thorough review process we were finally glad to accept 10 long papers and 5 short papers, the contents of this postworkshop proceedings. We hope that these contributions provide you with an overall impression about the state-of-the-art of object-oriented development of embedded RT software in industry as well as about still to be solved problems and ongoing research activities in this area.

We also hope that you enjoy reading these papers as much as all the participants of the two OMER workshops enjoyed the two events including a visit of the famous monastery ”Kloster Andechs”, one of the eldest Bavarian places of Pilgrimage and beer production.

Last but not least we would like to thank all persons, who spent a lot of time and efforts to make the two workshops such a success and who made it possible to publish the OMER proceedings: the authors, speakers, programme committee members, and participants of the two events, the reviewers of the contributions of this volume, the local staff of the ”Tagungsstätte des Bayerischen Bauernverbandes”, the local workshop organization team from the Institute of Software Technology at the University of the German Armed Forces, and the following sponsors of OMER-2: ARTiSAN Software, I-Logix Inc., Rational Software Corporation, and Telelogic AB.

The Program Co-Chairs

Peter P. Hofmann, DaimlerChrysler Research & Technology, Esslingen
Andy Schürr, Realtime System Lab, Darmstadt University of Technology

Program Committee of OMER workshop

M. von der Beeck, BMW, Munich, Germany

G. Engels, University of Paderborn, Germany

U. Epple, RWTH Aachen, Germany

M. Fuchs, BMW, Munich, Germany

E. Gery, I-Logix, USA

P. Hofmann, DaimlerChrysler, Esslingen, Germany

J. Kaiser, University of Ulm, Germany

R. Resch, Berner & Mattner GmbH, Munich, Germany

B. Rumpe, Munich University of Technology, Germany

S. Schmerler, 3S Engineering LLC, San Jose, USA

A. Schürr, Darmstadt University of Technology, Germany

H.-C. von der Wense, Motorola GmbH, Munich, Germany

U. Westermeier, Integrated Systems GmbH, Marburg, Germany

A. Zamperoni, Bosch Telecom, Frankfurt, Germany

Program Committee of OMER-2 workshop

M. von der Beeck, BMW, Munich, Germany

M. Cohen, I-Logix, USA

W. Damm, OFFIS, Oldenburg, Germany

G. Engels, University of Paderborn, Germany

P. Hofmann, DaimlerChrysler, Esslingen, Germany

D. Hogrefe, University of Lübeck, Germany

J. Hooman, University of Nijmwegen, Netherlands

J. Kaiser, University of Ulm, Germany

B. Møller-Pedersen, Ericsson AS, Billingstad, Sweden

U. Pansa, Telelogic, Germany

A. Radermacher, Siemens AG, Munich, Germany

R. Resch, Berner & Mattner GmbH, Munich, Germany

S. Schmerler, 3S Engineering LLC, San Jose, USA

A. Schürr, Darmstadt University of Technology, Germany

B. Selic, ObjecTime Limited, Kanata, Canada

A. Shaw, University of Washington, Seattle, USA

H.-C. von der Wense, Motorola GmbH, Munich, Germany

J. Ziegler, Nokia, Helsinki, Finland

Additional reviewers of Postworkshop Proceedings

M. Broy, Munich University of Technology, Germany

M. V. Cengarle, Munich University of Technology, Germany

J. H. Hausmann, University of Paderborn, Germany

A. Rausch, Munich University of Technology, Germany

S. Sauer, University of Paderborn, Germany

C. Schröder, Telelogic, Germany

M. Sihling, Munich University of Technology, Germany

Contents

Part I: Invited Talks and Technical Presentations

David Harel
On the Behavior of Complex Object-Oriented Systems 11

Bran Selic
Using UML to Model Complex Real-Time Architectures 16

Morgan Bjoerkander
UML and Real-time Systems 22

Andreas Korff
UML for Embedded Real-time Systems and the UML Extensions by ARTiSAN Software Tools 36

Alexei Filippov, Andrei Borshchev
Daimler-Chrysler Modeling Contest: Car Seat Model 46

Peter Braun, Oscar Slotosch
Development of a Car Seat: A Case Study using DOORS, AUTOFOCUS and the Validator 51

Part II: OO Development of Automotive Software

Ulrich Freund, Alexander Burst
Model-Based Design of ECU Software - A Component Based Approach 67

Jörg Petersen, Torsten Betram, Andreas Lapp, Kathrin Knorr, Pio Torre Flores, Jürgen Schirmer, Dieter Kraft, Wolfgang Hermsen
UML Meta Model Extensions for Specifying Functional Requirements of Mechatronic Components in Vehicles 84

Peter Braun, Martin Rappl
A Model-Based Approach for Automotive Software Development 100

Part III: OO Development of Production Control and Automation Systems

Holger Giese, Ulrich A. Nickel
Towards Service-Based Flexible Production Control Systems and their Modular Modeling and Simulation 106

Torsten Heverhagen, Rudolf Tracht
Implementing Function Block Adapters 122

Andreas Metzger, Stefan Queins
Specifying Building Automation Systems with PROBanD, a Method Based on Prototyping, Reuse, and Object-orientation 135

Part IV: OO Development of Realtime Systems

Jorge L. Diaz-Herrera, Hanmei Chen, Rukshana Alam
An Isomorphic Mapping for SpecC In UML 141

Theodor Tempelmeier
On The Real Value Of New Paradigms 157

Dominikus Herzberg, Andre Marburger
State Machine Modeling: From Synch States to Synchronized State Machines 175

On the Behavior of Complex Object-Oriented Systems*

David Harel
The Weizman Institute of Science, Rehovot, Israel
and
I-Logix, Inc., Andover, MA

Over the years, the main approaches to high-level system modeling have been structured-analysis and object-orientation (OO). The two are about a decade apart in initial conception and evolution. SA started out in the late 1970's by De Marco, Yourdon and others, and is based on "lifted" classical procedural programming concepts up to the modeling level and using diagrams [CY79]. The result calls for modeling system structure by functional decomposition and the flow of information, depicted by hierarchical data-flow diagrams. As to system behavior, the mid 1980's saw several methodology teams (such as Ward/Mellor [WM85], Hatley/Pribhai [HP87] and our own Statemate team [HHN+90]) making detailed recommendations enriching the basic SA model with means for capturing behavior based on state diagrams, or the richer language of statecharts [Har87]. A state diagram or statechart is associated with each function to describe its behavior. Carefully defined behavioral modeling, we should add, is especially crucial for embedded, reactive, and real-time systems. A detailed description of the way this is done in the SA framework appears in [HM98]. The first tool to enable model executability and code synthesis of high-level models was Statemate, made commercially available in 1987 (see [HHN+90], [Inc]).

OO modeling started in the late 1980's. Here too, the basic idea for system structure was to "lift" concepts from object-oriented programming up to the modeling level, and to do things with diagrams. Thus, the basic structural model for objects in Booch's method [Boo94], in OMT [RBP+91], in the ROOM method [SGW94], and in many others (e.g., [CD94]), has notation for classes and instances, relationships and roles, and aggregation and inheritance. Visuality is achieved by basing this model on an enriched form of entity-relationship diagrams. As to systems behavior, most OO modeling approaches, including those just listed, adopted the statechart language for this. A statechart is associated with each class, and its role is to describe the behavior of the instance objects.

However, here are subtle and complicated connections between structure and behavior, that do not show up in the simpler SA paradigm. Here classes represent dynamically changed collections of concrete objects, and behavioral modeling must address issues related to their creation and destructions, the delegation of messages, the modification and maintenance of relationships, aggregation, true inheritance, etc. These issues were treated by OO methodologists in a broad spectrum of degrees in detail — from vastly insufficient to adequate. The test, of course, is whether the languages for structure and behavior and their

*Reprint of abstract from *OMER Workshop Proceedings, Peter Hofmann and Andy Schüerr (eds.), Bericht Nr. 1999-01, University of the Federal Armed Forces Munich, Neubiberg, Germany*

inter-links are defined sufficiently well to allow full model execution and code synthesis. This has been achieved only in a couple of cases, namely in the ObjecTime tool (based on the ROOM method of [SGW94]), and in the Rhapsody tool. Rhapsody (see [Inc]) is based on the executable modeling work presented in [HG97], which was originally intended as a carefully worked out language set based on Booch and OMT object model diagrams driven by statecharts, and addressing the issues above in a way sufficient to lead to executability and full code synthesis.

In a remarkable departure from the similarity in evolution between the SA and OO paradigms for system modeling, the last three years have seen OO methodologists working together. They have compared notes, have debated the issues, and have finally cooperated in formulating the UML, which was adopted in 1997 as a standard by the OMG (see [Cor]). This sweeping effort, which in its teamwork is reminiscent of the Algol'60 and Ada efforts, has taken place under the auspices of Rational Corp., spearheaded by Booch, Rumbough and Jacobson. Version 0.8 of the UML was released in 1996 and was a rather open-ended and vague, lacking in detail and well thought-out semantics. For about a year, the UML team went into overdrive, with a lot of help from methodologists and language designers from outside Rational Corp. Out team contributed quite a bit too, and the languages underlying Rhapsody [HG97], [Inc] are indeed the executable kernel of the UML. The version of the UML adopted by the OMG is thus much tighter and more solid than version 0.8. With some more work there is a good chance that the UML will become not just an officially approved standard, but the main modeling mechanism for the software that is constructed according to the object-oriented doctrine. And this is no small matter, as more and more software engineers are now claiming that more and more kinds of software are best developed in an OO fashion.

The recent wave of popularity that the UML is enjoying will bring with it not only the UML books written by Rational Corp. authors (see, e.g., [RJB99]), but a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra-careful in finding the really worthy trees in this forest. Despite this, one must remember that right now UML is a little *too* massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth as to make clear their relationships with the constructive core of UML (the class diagrams and the statecharts). Moreover, there are still major problems in the general area of behavioral specification and design of complex object-oriented systems that await treatment. These still require extensive research.

Here are brief discussions of two examples of research directions that seem to me to be extremely important. One has to do with message sequence charts (MSC's) and their relationship with state-based specification, and the other has to do with inheriting behavior.

As to the first one, there is a dire need for a highly expressive MSC language, with a clearly defined graphical syntax and a fully worked out formal semantics. Such a language is needed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios. It is also a prerequisite to a thorough investigation of what might be the problem in object-oriented specification. The former is what engineers will typically do in the early stages of behavioral modeling; namely, they come up with use-cases and the scenarios that capture them, specifying the inter-relationships

between the processes and object instances in a linear or quasi-linear fashion of temporal progress. That is, they come up with the description of the scenarios, or "stories" that the system will support, each one involving all the relevant instances. A language for scenarios is best used for this. The latter, on the other hand, is what we would like the final stages of behavioral modeling to end up with; namely, a complete description of the behavior of each of the instances under all possible conditions and in all possible "stories". For this, a state-machine language such as statecharts appears to be most useful. The reason the state-machine intra-object model is what we want as an output from design stage is for implementation purposes: ultimately, the final software will consist of code for each process or object. These pieces of code, one for each process or object instance, must — together — support the scenarios as specified in the MSC's. Thus the "all relevant parts of the stories for one object" descriptions must implement the "one story for all relevant objects" descriptions.

Now, there are several versions of MSC's, including the ITU standard (see [ITU96]), and the UML also has a version of sequence diagrams as part of its language. However, both versions are extremely weak in expressive power, being based essentially on simple constraints on the partial order of events. Nothing much can be specified about what the system will actually do when run. A particular troublesome issue is the need to be able to specify "no-go" scenarios, ones that are not allowed to occur. In short, there is a serious need for a more powerful language for sequences. In a recent paper [DH99], we have addressed this need, proposing an extension of MSC's, which we call *live sequence charts* (or *LSCs*). One of the main extensions deals with specifying *"liveness"*, i.e., things that must occur. LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart, and locally, when specifying events, conditions and progress over time within a chart. (In doing so it makes possible the natural specification of forbidden behavior.) LSCs also support subcharts, synchronization, branching and iteration. It is far from clear whether this language is exactly what is needed: more work on it is required, experience in working with it, and of course an implementation. Nevertheless, it does make it possible to start looking seriously at the two-way relationship between the arfomentioned dual views of behavioral description. How to address this *grand dichotomy of reactive behavior*, as we like to call it, os a major problem. For example, how can we synthesize a good first approximation of the statecharts from LSCs? Finding efficient ways to do this would constitute a significant advance in the automation and reliability of system development. In very recent work, as of yet unpublished, we propose a first-cut at devising such synthesis algorithms and at analyzing their complexity [HK99a].

The second direction of research involves inheriting behavior. Inheritance is one of the key topics in the object-oriented paradigm, but when working with analysis and design levels (rather than in the programming stage) it is not at all clear what exactly it means for an object of type *B* to be also an object of the more general type *A*. In virtually all approaches to inheritance in the literature, the **is-a** relationship between classes *A* and *B* entails a basic minimal requirement of *protocol conformity*, or subtyping, which roughly means that is should be possible to "plug-in" a *B* wherever an *A* could have been used, by requiring that what can be requested of B is consistent with that can be requested of *A*.

In addition, *structural conformity*, or subclassing, is often requested, to the effect that *B*'s internal structure, such as its set of composites and aggregates, is consistent with that of *A*.

Nevertheless, these form only weak kinds of subtyping, and they say little about the *behavioral conformity* of *A* and *B*. They require only that the plugging in be possible without causing incompatibility, but nothing is guaranteed about the way *B* will actually operate when it replaces *A*. Thus we don't have full behavioral substitutability, but merely a form of consistency. In fact, *B*'s response to an event or an operation invocation might be totally different from *A*'s. Here we are concerned with investigating the plausibility (and indeed also the very wisdom) of guaranteeing full behavioral conformity. In practice, behavioral conformity is often too stringent; may times one does not expect the inheritance relationship between *A* and *B* to mean that anything *A* can do *B* can do and in the very same way. They are often satisfied with guaranteeing that anything *A* can do, *B* can be *asked* to do, and will look like it is doing, but it might do so differently and produce different results.

In recent work, also not yet published [HK99b], we have obtained preliminary results that show that on a suitable schematic, propositional-like level of discourse there are strong connections between questions of inheritance and well-known semantic notions of refinement between specifications (such as trace containment and simulation). We also have several results about the computational complexity of detecting and enforcing behavioral conformity. However, here too there is still much research to be done, including the discovery of restrictions on behavioral specification that would guarantee behavioral conformity, and algorithms for finding out if given models satisfy such restrictions.

Many other significant challenges remain, for which only the surface has been scratched. Examples include true formal verification of software modeled using high-level visual formalisms, automatic eye-pleasing and structure-enhancing layout of the diagrams in such formalisms, satisfactory ways of dealing with hybrid object-oriented systems that involve discrete as well as continuous parts, and much more.

It is probably no great exaggeration to say that there is a lot more that we *don't* know and *can't* achieve yet in business than what we do know and can achieve. Still, the efforts of scores of researchers, methodologists and language designers have resulted in a lot more than we could have hoped ten years ago, and for this we should be thankful and humble.

References

[Boo94] G. Booch. *Object-Oriented Analysis and Design, with Applications*. Benjamin/Cummings, 2nd edition, 1994.

[CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling With Syntropy*. Prentice Hall, New York, 1994.

[Cor] Rational Corp. document on the UML. http://www.rational.com/uml/index.html.

[CY79] L.L. Constantine and E. Yourdon. *Structured Design*. Prentice Hall, Englewood Cliffs, 1979.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R.Gorrieri, editors, *Proc. 3er IFIP Conf. on Formal Methods for Open Object-based Distributed Systems*, pages 293–312. Kluwer Academic Publishers, 1999.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Prog.*, 8:231–274, 1987.

[HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, pages 31–42, July 1997.

[HHN+90] D. Harel, H.Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Trans. Soft. Eng.*, 16:396–406, 1990.

[HK99a] D. Harel and H. Kugler. manuscript in preparation. 1999.

[HK99b] D. Harel and O. Kupferman. manuscript in preparation. 1999.

[HM98] D. Harel and M.Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[HP87] D. Hatley and I. Pribhai. *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.

[Inc] I-Logix Inc. products web page. http://www.ilogix.com/fs_prod.htm.

[ITU96] ITU, Geneva. *ITU-TS Recommendation Z.120: Message Sequence Charts (MSC)*, 1996.

[RBP+91] J. Rumbough, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RJB99] J. Rumbough, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.

[WM85] P. Ward and S. Mellor. *Structured Development fro Real-Time Systems*, volume 1, 2, 3. Yourdon Press, New York, 1985.

Using UML to Model Complex Real-Time Architectures*

Bran Selic
ObjecTime Limited
Kanata, Ontario, CANADA

1 Introduction

The term "software architecture" is widely used but infrequently specified. We define it as: *the organization of significant software components interacting through interfaces, those components being composed of successively smaller components and interfaces*[1].

Note that, when dealing with architectures, we are only interested in *high-level* features. Architecture necessarily involves abstraction — the elimination of irrelevant detail so that we can better cope with complexity.

The term "organization" pertains to the high-level structure of a system: its decomposition into parts and their mutual relationships (interaction channels, hierarchical containment, etc.). However, architecture is more than just structure. It includes rules on how system functionality is achieved across the structure. This includes high-level end-to-end behavioral sequences by which a system realizes its use cases.

Structure and behavior come together in interfaces, which, appropriately, also play a fundamental role in the specification of a software architecture. The last part of the definition above is simply a recursion of the first part. This tells us that architecture is a relative concept. It does not just occur at the "top" level of decomposition. A system may be decomposed into a set of subsystems, each of which, in turn, can be viewed as a system in its own right etc.

Software architectures play pivotal role in two types of situations. During initial development of a software system, an architecture is used to separate responsibilities and distribute work across multiple development teams. If the architecture is well defined, it should be straightforward to put the individually developed parts together and complete the system. Unfortunately, properly specified architectures are rare, which explains why there is a large incidence of so-called "integration" problems. (Almost invariably, an integration problem is the result either of an inadequate architecture or an inadequately specified architecture.)

*Reprint of abstract from *OMER Workshop Proceedings, Peter Hofmann and Andy Schüerr (eds.), Bericht Nr. 1999-01, University of the Federal Armed Forces Munich, Neubiberg, Germany*

[1] I owe this definition to Grady Booch

However, even more importantly, software architectures play a crucial role in system evolution. It can be safely said that the architecture is the fundamental determinant of a system's capacity to undergo evolutionary change. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized for just its initial set of requirements.

2 Requirements for Architectural Modeling

Complex systems often have complex architectures. An architectural modeling language must have the range to define all the necessary elements of that in a clear and unambiguous manner.

Most complex software systems are structured in some layered fashion. Layering is one of the most common ways of dealing with complexity. Yet, no standard programming language used in industry provides the concept of a "layer" as a first class concept. Instead, layering is simulated in a variety of ways, such as the use of compilation module boundaries, none of which are adequate to precisely capture the subtle semantics of layering. For instance, although layered architectures are often depicted as simple "onion-peel" structures, the layering in most real-time systems is far more complicated. Consider, for example, the well-known seven-layer architecture of ISO's Open System Interconnection standard. This architecture ideals exclusively with communication aspects of a system. It is typically implemented as an application on top of an operating system, which represents an orthogonal layer hierarchy. In other words, complex systems require multiple dimensions of layering — two are never enough.

Another issue associated with architectures is the potential for reuse. Many modern systems are built around the "product line" concept. A product line is a set of different products that are built around a common abstract architecture. This architecture is then refined in different ways to realize individual products. This leads us to the requirement for subclassing at the architectural level. Needless to say, with very poor support for high-level concepts such as layering, no standard programming language in use today supports such a capability.

It is clear that we need a new breed of specification languages that is capable of addressing these requirements.

3 An Architectural Modeling Language

We define an architectural modeling language building on the ideas of the ROOM modeling language [SGW94]. A part of ROOM was specifically designed for modeling architectures of complex real-time systems. However, we will expand on those ideas and, furthermore, express them in the industry standard Unified Modeling Language (UML) [BRJ98b] [BRJ98a] [OMG97b] [OMG97a]. This allows us to take advantage of seman-

tics and notation that are widely recognized by software practitioners. Specifically, we use the extensibility mechanisms[2] of UML: stereotypes, tagged values, and constraints. In other words, we define our specific architectural modeling concepts as specializations of generic UML concepts. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints.

3.1 Capsules

The basic concept is that of an architectural object called a *capsule*. A capsule is a stereotype of the UML class concept with some specific features:

- it is always an active object, which means that it has behavior that executes concurrently with other behaviors
- it has an encapsulation shell such that it not only hides the implementation from outside observers, but also prevents the implementation from directly (and arbitrarily) accessing the external environment; in other words, it is an encapsulation shell that works in both directions [3]
- it may be a truly distributed object — it may even span multiple physical nodes; this makes it suitable for modeling physically distributed conceptional entities.

Capsules are used to capture major architectural components of real-time systems. For instance, a capsule may be used to capture an entire subsystem, or even a complete system. As we shall see later on, a capsule may encapsulate any number of internal capsules.

One of the interesting features of capsules is that they can have multiple interfaces, called *ports*. Each interface presents a distinct aspect of a capsule. Different collaborators can access different interfaces, possibly in parallel. We shall describe ports in more detail in the following section.

A capsule uses its port for *all* interactions with its environment. The communication is done using signals, which can be used to carry both synchronous and asynchronous interactions. The advantage of signals as the underlying communication vehicle is that, in contrast to communications based on procedure calls, they are more amenable to distribution.

Because capsules are a kind-of class, they can be subclassed. This gives us the capability to produce variations and refinements of architectural components and even entire architectures.

[2]The term “extensibility mechanism” is somewhat misleading since they are really used not to introduce new concepts, but to allow the definition of specialized versions of existing ones.

[3]In classical OO languages, this is not the case: while the implementation is hidden from external entities, the implementation has unhindered access to the environment. This creates problems since a “component” in some class library may be very tightly coupled with other entities. Unfortunately, this coupling can only be determined by careful inspection of the implementation.

3.2 Ports

In contrast to interfaces one finds in traditional object-oriented programming languages, ports are distinct objects (stereotypes of the UML class concept). They convey signals between the environment and the capsule. The types of signals and the order in which they may appear is defined by the *protocol* associated with the port. Protocols are discussed in the next section.

Ports are used not only for receiving incoming signals, but also for sending all outgoing signals. This means that a capsule is fully encapsulated: internal components never reference an external entity, but only communicate through ports. Consequently, a capsule class is fully self contained — which makes it a truly reusable component.

3.3 Protocols

A *protocol* specifies a set of valid behaviors (signal exchanges) between two or more collaborating capsules. However, to make such a dynamic pattern reusable, protocols are de-coupled from a particular context of collaborating capsules and are defined instead in terms of abstract entities called *protocol roles*. A protocol role represents the activities of one participant of a protocol. Formally, it is defined by a set of incoming signals, a set of outgoing signals, and an optional behavior specification. The behavior specification represents that subset of the behavior specified for the overall protocol that directly involves this role.

A particularly common type of protocols are *binary* protocols, which involve only two roles. For binary protocols it is sufficient to define one protocol role to define the entire protocol.

The relationship between ports and protocols is crucial: each port plays exactly one role in some protocol. The protocol role of a port represents the type of the port.

Protocol are modeled in UML as a stereotype of the collaboration concept. Since collaborations are generalizable elements, they too can be refined using inheritance-like mechanisms to produce variations and refinements.

Protocol roles are stereotypes of the classifier role concept in UML.

3.4 Connectors

To allow capsules to be combined into aggregates, we define the concept of a *connector*. A connector is an abstraction of a message-passing channel that connects two or more ports. Each connector is typed by a protocol that defines the possible interactions that can take place across that connector.

Connectors are stereotypes of the association class concept in UML with the added constraints that they can only pass signals and that their behavior is governed by a protocol.

3.5 Composite Capsules

Using the simple concepts of capsules, ports, and connectors, we can easily assemble complex aggregates of diverse capsules that achieve complex functionality. The design paradigm behind this is very similar to the design of hardware. Complex systems emerge by interconnecting simpler specialized parts drawn from a components catalog.

It is often the case that we may need to reuse a particular object composition pattern in a variety of different situations. In other words, we would like to make these assemblies into components in their own right. For this purpose, the object paradigm gives us the class concept. A particularly convenient way of realizing such a class is to define it as a capsule. This makes the capsule concept recursive: a capsule can be decomposed into lesser capsules, an so on. There are no theoretical limits to this hierarchy, capsules can be nested to whatever extent is necessary to realize our desired system. Note that such composite capsules can have their own ports, like any other capsules. These ports may be used to delegate functionality, using an internal connector, to some internal component. Such ports are called *relay* ports since their function is simply to act as a funnel for signal traffic. Alternatively, a port may be connected directly to a state machine. This type of port maintains a queue of incoming signals that have been received but not yet processed by the state machine. These ports are called *end* ports.

A composite capsule, therefore, represents a network of collaborating capsules. A particularly important feature of composite capsules is their *assertion semantics*. What this means is that when a composite capsule is instantiated, all of its internal nested parts are *automatically* instantiated along with it. Furthermore, since the internal capsules may themselves be composites, an entire system can be created with just one single “create” action. The consequence of assertion semantics is that the designer is liberated from the often tedious and highly error prone task of having to write the explicit code that generates the complex internal structure piece by piece. This not only saves effort, but also provides a tremendous boost in reliability, because in many dynamic systems, the code used to establish a structure can represent a major percentage of the overall software.

4 Summary

Architecture plays a key role in the design of complex real-time systems. In order to specify the types of architectures that are common in this domain, we need a suitable set of modeling concepts. We have presented here a very simple set of modeling concepts consisting of capsules, ports, protocols, and connectors, that has already been proven effective in industrial practice. To make these modeling capabilities available to the broadest set of users possible and to take advantage of the commercial tools, we have expressed them

using the industry standard Unified Modeling Language. For this purpose, we have used the extensibility mechanisms of UML. These allowed us to very effectively capture the full semantics of the concepts in a manner consistent with the general semantics of UML.

5 Acknowledgment

The author recognizes the major contribution of Jim Rumbough of Rational Software Corporation who collaborated very closely with the author on the work reported here.

References

[BRJ98a] G. Booch, J. Rumbough, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley Publishing Co., 1998.

[BRJ98b] G. Booch, J. Rumbough, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Publishing Co., 1998.

[OMG97a] OMG. *UML Notation Guide (Version 1.1)*. The Object Management Group, Framington MA, 1997. Doc. ad/97-08-05.

[OMG97b] OMG. *UML Semantics (Version 1.1)*. The Object Management Group, Framington MA, 1997. Doc. ad/97-08-04.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.

[SR98] B. Selic and J. Rumbough. Using UML for Modeling Complex Real-Time Systems. ObjecTime Limited/Rational Software Corp. white paper, March 1998.

UML and Real-time Systems

Morgan Björkander

Telelogic AB
PO Box 4128
SE-203 12 Malmö
Sweden
mbj@telelogic.com

Abstract. UML has traditionally been used to specify object-oriented software systems. With its rising popularity, the desire to use it for various vertical domains have grown stronger, and in this paper we focus on requirements from the real-time domain. In particular, we look at how tools and features from the real-time domain have affected the standardization efforts when further developing the next generation of UML, called UML 2.0. As part of the language proper, the primary concern is to cover soft real-time aspects, while hard real-time aspects are handled as part of the Real-Time UML profile, which focuses on mechanisms to support schedulability and performance analysis. This paper focuses on the former aspects, but also touches on the latter aspects. In addition, we examine some of the influences from languages that are normally associated with real-time, such as SDL and UML-RT.

1. Introduction

UML was created specifically to deal with the specification, visualization, construction, and documentation of software, and had a significant object-oriented slant, to the extent that the term ”C++ in pictures” was coined. Its ever increasing popularity and inherent flexibility have caused the language to transcend its original boundaries set by the object-oriented paradigm, and it is now used in a wide array of settings. However, the language often lacks provisions to deal with concepts or constructs of certain domains, and one of the areas where this was noted early on is the real-time domain. To some extent, this lack can be managed through the use of the built-in extensibility mechanism of UML, which allows the creation of profiles that adapt the language for specific needs.

Since its adoption over five years ago, both vendors and users have gained significant experience with the language, and we now know where to look for useful concepts and also which constructs did not quite pan out as intended. In addition, new emerging technologies—such as component-based development—could not always be satisfactorily captured by the existing UML, and for these reasons a revision process was initiated by the OMG in 2000 with the intent to create a new major revision of UML called UML 2.0. A number of deficiencies were identified, and solutions in the form of proposals were solicited. At this point in time the revision process is nearing completion, and the expected outcome can be assessed.

The focus of this paper is soft real-time systems, by which we mean the ability to express event-driven, distributed systems, where asynchronous communication and concurrent execution are important factors, and while some of the constructs described here originate in the telecom industry, they are nowadays commonly occurring in for example the automotive and aerospace industries, and are expected to permeate the way systems in general are modeled.

2. The Object Management Group and UML

The Object Management Group (OMG) is the body that is responsible for developing and maintaining UML and other related technologies, and it is entirely driven by its members. The Unified Modeling Language (UML) was first adopted in 1997 as UML 1.1, and several minor revisions have since been adopted, the latest being UML 1.4 [OMG01]. These minor revisions have essentially been bug fixes, since the OMG process prohibits larger changes to existing adopted technologies in order to protect tool implementations. Significant changes to UML can only be done through a major revision process, and this was initiated in 2000 when four Request for Proposals (RFPs) related to UML 2.0 were issued:

- UML 2.0 Infrastructure RFP [OMG00b]: Define the elements that are used to specify metamodels like UML, and also the mechanisms used to extend metamodels.
- UML 2.0 Superstructure RFP [OMG00d]: Define the elements that are used to specify structure and behavior of a system. This is what we normally call UML, and includes the definition of all diagrams such as class diagrams, sequence diagrams, etc.
- UML 2.0 Diagram Interchange RFP [OMG00a]: Define the rules for how models and diagrams are interchanged between tools.
- UML 2.0 OCL RFP [OMG00c]: Define a language for specifying constraints, which is used for example to more precisely pin down the semantics of UML.

Each member company of the OMG is free to enter a proposal, called a submission, to any issued RFP, and normally several companies create joint submissions to strengthen their positions, and examples of such submissions can be found in [U2P02a] and [UU02]. Some of the examples that are shown in this paper are based on one of the submissions to the UML 2.0 Superstructure RFP created by the U2 Partners [U2P02b], which includes the following companies comprised of both vendors and users: Alcatel, CA, Enea, Ericsson, HP, I-Logix, IBM, IONA, Kabira, Motorola, Oracle, Rational, Softeam, Telelogic, Unisys, and Webgain. All examples come with the caveat that the submissions are not yet finalized, and notation and semantics that are described in this paper may differ from the final outcome.

Since UML is a general-purpose modeling language, there are several real-time issues that are not covered directly by the language. Some of these have been included in UML 2.0, as described below, while others are covered in accompanying profiles. One real-time profile has already been adopted, and deals with modeling of schedulability, performance, and time [OMG02c]. Another real-time profile dealing with modeling of quality of service and fault tolerance using UML was initiated when the RFP was issued earlier this year [OMG02b] and is currently in the works.

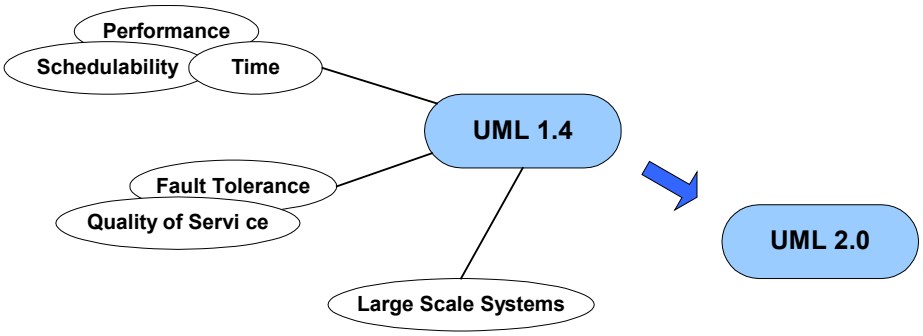


Fig. 1. The original roadmap for the real-time work within the OMG included work to cover modeling of large-scale systems; however, this part was entirely subsumed by the work on UML 2.0. Note that the profiles that are adopted work together with UML 1.4, and need to be updated to be usable with UML 2.0.

An adopted technology that will also play a significant role in UML 2.0 is the one that deals with an action semantics for the UML [OMG02a], and gives the basic capabilities needed to create executable models in UML, which for example enables performance simulation when combined with the real-time profiles. Of course, executable models have other more important effects, some of which we cover below.

3. UML Tool Support

Tool support for UML comes in different forms, but traditionally there are two approaches that clearly dominate the market when it comes to building applications from the models:

- Round-trip engineering
- Code-insertion

Using the *round-trip engineering* approach, stub-code in a given programming language such as C++ or Java is generated from the model based on a set of mapping rules, detailing how classes, attributes, operations, etc. from the model are to be represented in code. The generated code normally lacks functionality and detailed behavior, and these things need to be added manually to the code. While updating the code, it is important to make sure that the model and the code remain consistent with each other, which is accomplished by synchronizing the code and the model.

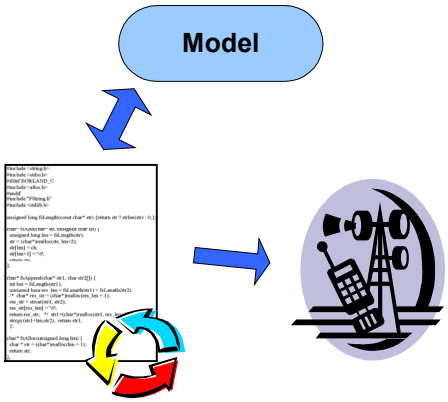


Fig. 2. Round-trip engineering implies that stub-code is automatically generated, manually manipulated in some way, and then the changes may be synchronized with the model to the extent that the modeling language is capable of representing the changes. Typically, the final application is then compiled from the code.

What characterizes this approach is that the "code is king", and one of the risks is that since everything revolves around the code the model is often treated as an afterthought used only for documentation purposes. A consequence of this is that the synchronization part is often sacrificed towards the end of a project when deadlines are getting tight, meaning that the model and the code will slowly but surely diverge, eventually to the point where synchronization is no longer meaningful; reverse engineering of the code will do just as well. However, if this happens, there is something seriously wrong with the development process, and maintenance of the system will likely be problematic.

Another drawback, albeit not as serious, is that you have to know what programming language to use together with the modeling language, and if at one point it becomes necessary to change programming language, most of the coding needs to be redone.

The synchronization problem of the round-trip engineering approach is solved using the *code insertion* approach, where code fragments are added directly to the model, usually in the form of detailed behavior. This way, pieces of code in a particular programming language are scattered throughout the model, and while this may be detrimental to clarity it forces the model to always be up to date if the goal is to be able to create an application. The code that is generated using this approach should not have to be manually updated, since all the behavior of the system is embedded in the model.

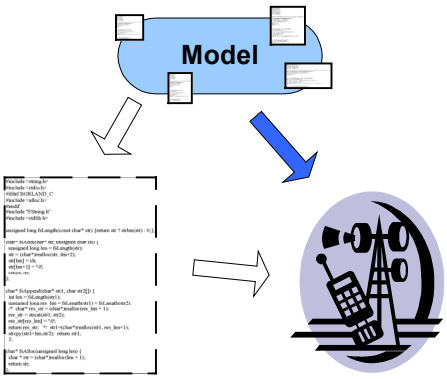


Fig. 3. Pieces of code are written directly in the model in the code insertion approach. Usually, this means that code specific to some programming language is attached for example to a transition to indicate the actions that should occur between two states. Conceptually, the final application is then built from the model, while in practice it is compiled from generated code.

This approach does suffer from the same problem as the round-trip engineering approach in that it is bound to a particular programming language, but the problem is aggravated by the fact that the model and the code is mixed with each other. In some ways, this is similar to embedding assembly code in C code, but it is not done for the same efficiency reasons.

When upgrading to UML 2.0 it is expected that both approaches will continue to hold their own in slightly modified forms. The round-trip engineering approach can be improved through better tool support, while the code insertion approach naturally evolves in such a way that detailed behavior can be expressed entirely in the model, and there no longer is any need to mix code and model. The latter approach is further elaborated below.

4. Building Large-Scale Systems

Many of the ideas that get incorporated into the UML come from tool extensions to the existing UML standard or from other modeling languages. As was stated earlier, the roadmap within the OMG to support modeling of real-time systems at one point incorporated modeling of large-scale systems, and this was based on for example SDL [ITU00] and UML-RT [SR98]. However, due to the flexible and generic way these concepts are defined, it was deemed more appropriate to define the necessary constructs directly in UML 2.0 instead of relying on a profile.

Using building blocks to create structure

Scalability is one of the keywords when designing large and complex systems, and a component-based approach is key to dividing a problem into manageable chunks. Both SDL and UML-RT are built around the concept of a building block, called *agent* (which

includes the concepts *block* and *process*) and *capsule*, respectively. These building blocks are also natural distribution units that execute with their own thread of control. Such a building block can be used in two ways. It can be hierarchically decomposed into smaller and smaller building blocks until only very trivial ones remain in a top-down fashion, or a number of building blocks can be put together into larger building blocks to provide the desired functionality in a bottom-up fashion. This way, a building block may consist of as many layers as are practical.

The purpose of a building block is to encapsulate the behavior and structure that make up its implementation, i.e., to hide unnecessary detail from anyone who needs to use the building block in some way. An internal structure of a building block can be provided through other building blocks that are connected with each other, while the behavior of a building block may be provided for example through a state machine. From the outside, a building block is viewed as a black box whose interfaces are clearly defined and provide users of the building block with enough information about its services to be able to use it.

The designer of a building block views it as a white box, i.e., the gory details about the internals of the building block are known. A building block may have both structure and behavior, in which case the behavior is often used to control the structure. Note that the white box view typically only comprises one layer, since the building blocks that are connected together to form the internal structure are viewed as black boxes. These may in turn have internal structure and behavior of their own. By "zooming" into a black box you get its white box view, and by "zooming" out of a white box you get its black box view.

In UML 2.0, the building block concept is represented through structured *classes*, which may have internal structure and behavior. While not strictly necessary, it is common for structured classes to be active, i.e., to have their own thread of control. It should be noted that it is still possible to use classes in pretty much the same way as in UML 1.4; in this paper, however, we focus on features that have been added to the language to better support real-time systems development. Graphically, active classes are denoted using vertical bars on the sides of the class.

Interface-based design

One of the main ideas behind using building blocks is that each one is a self-contained entity that can be reused in many contexts, which makes it very important to properly define the interfaces of a building block.

In UML 1.4, provisions are made to show that a class realizes an interface, and graphically this is normally shown using a lollipop symbol attached to the class. In order to show how it interacts with other entities, without knowing what those entities are, UML 2.0 allows the definition of not only *provided interfaces* but also of *required interfaces*, i.e., the interfaces or services that others must provide in order for the class to function correctly. Graphically, a required interface is shown using a symbol that resembles the lollipop symbol, but where the circle is replaced with a half-circle. An interface may additionally or alternatively be shown using a class-like notation, which allows its attributes and operations to be shown. It should be noted that an interface itself is neither required nor provided; it is only the relationship between the class and the

interface that determines how it is used. If the relationship is an implementation, the interface is provided, and if the relationship is a usage, the interface is required.

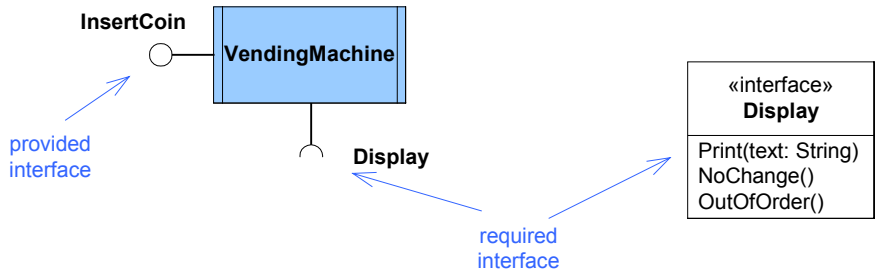


Fig. 4. A class may have provided and required interfaces that indicate the services realized by the class and the services that are expected by others. This gives the capability to develop each class as a stand-alone entity, where the interfaces through which the class is going to interact is the only necessary information about its environment. (Note that the attribute and operation compartments of the active class VendingMachine are elided.)

Interfaces expose the services that are provided by a class, and are the primary means by which a class encapsulates its implementation. *Interfaces* in SDL are treated in the same way as interfaces in UML 2.0, but use arrows instead of lollipops as the notation. In UML-RT, the corresponding mechanisms are *protocols* and *protocol roles*, and these can be built on top of the UML 2.0 interfaces.

Class communication and interaction points

When dealing with building blocks as the ones previously described, the normal behavior is that only classes that have matching interfaces are allowed to communicate with each other; a class that has a required interface can thus only communicate with another element (not necessarily a class) that provides the same interface, or a specialized interface. This way it is easy to specify contracts between different parties of a system, and also makes it simple for tools to prevent classes that don't have matching interfaces to be connected with each other.

In both SDL and UML-RT the concept of an interaction point plays a prominent role, and is called *gate* and *port*, respectively. In UML 2.0, the corresponding concept is called a *port*, which is simply typed by an interface and can be either required or provided. This works slightly different from the case where we did not have ports, but the underlying principle is the same. A port sits on the boundary of a class, and can be viewed either from inside the class (white box view) or from outside the class (black box view). In the former case it represents a view of the environment of the class, and in the latter case it represents a view of the class as seen from the environment. The primary purpose of a class, however, is to act as a connection point when classes are connected to each other, as is shown below. A class may further be addressed through any of its ports.

A composite port comprises a collection of required and/or provided ports, and is used to model when a port is typed by multiple interfaces or when a port should support bi-directional communication. The latter occurs when a composite port has both a

required and a provided port. Ports are usually named by their typing interfaces, but composite ports have to be given a name of their own.

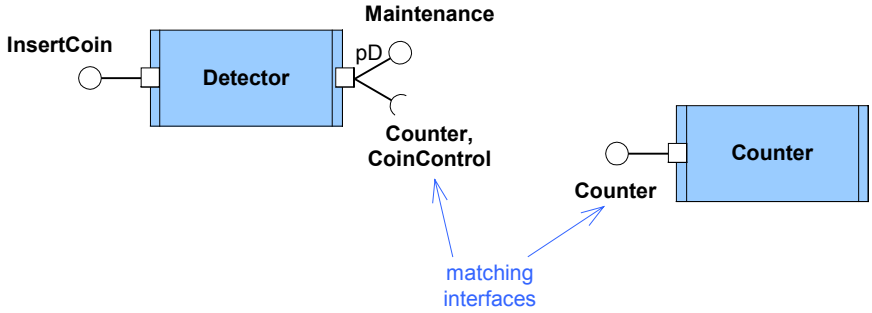


Fig. 5. Classes need matching required and provided interfaces in order to be able to communicate with each other. In order to deal with complex systems, it is possible to define interaction points—ports—that provide addressable viewpoints of a class.

One way to conceptually view a port is to consider it as a hole in the encapsulation of a class through which it is possible to look into the class, and all you can see is what is provided by the interfaces that type the port. (Likewise, it allows someone inside the class to look out at the environment in a similar manner.)

Connecting classes in an internal structure

The internal structure of a (containing) class defines how a number of classes communicate with each other in a specific context. In a traditional object-oriented approach classes are simply tied to each other through associations, and that's that. However, this means that an association is always applicable in any context where the class is used, and this is not desirable in a more component-based approach. Instead, you want to be able to express when there should be a connection between two classes depending on the context in which they are used and which interfaces they support.

Both SDL and UML-RT allow you to decompose agents and capsules into internal structures. In SDL, the internal structure is made up of a number of *instance sets,* whose gates may be connected through *channels*, whereas UML-RT relies on collaborations where the ports of *subcapsules* may be connected through *connectors*.

In UML 2.0, a *part* represents one or more "instances to come" and corresponds to an instance set or a subcapsule. A part is thus a specific usage of a class in an internal structure, and it is possible to have several different parts of the same class. Parts may be connected to each other through the use of *connectors*, and typically the connector is tied to a specific port of a part. Graphically, a part is shown using a rectangular symbol with the (optional) name of the part and the (mandatory) name of the used class separated by a colon.

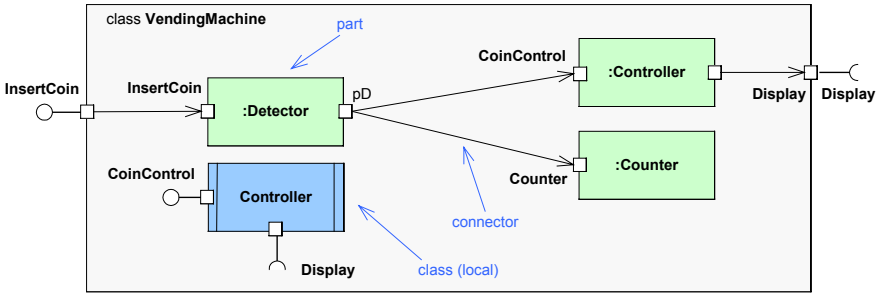


Fig. 6. A class can be used as part of an internal structure, and also be connected to other parts (of the same or another class). The parts and connections are only applicable in the context of the containing class. In previous figures the classes VendingMachine, Detector, and Counter have been defined, and here we look at the internal structure of the VendingMachine, where we use the classes Detector and Counter as parts that are connected to each other. In addition, there is a locally defined class Controller that is also used as a part. The provided interface Maintenance of the Detector is not used in this context. Note that the class VendingMachine can be used as part in another context.

There is a lifecycle dependency between a containing class and its internal structure in that when an instance of a containing class is created, instances of the classes that are represented through parts are also created at the same time. Likewise, when the instance of the containing class is terminated, the contained instances are also terminated. It is possible to give a multiplicity to the part, to indicate the number of instances to be created when the containing class is created and also to indicate the maximum number of instances that may exist at a time.

The behavior of a class

Earlier, it was mentioned that it is possible to mix behavior and structure as part of the internal structure. This is also reflected among the ports of a class, where behavioral ports that connect directly to the behavior of the class are distinguished from ports that connect to the parts of the class.

Normally, the behavior of an active class is expressed through a state machine, but it is also possible to use for example an activity. Because of the lifecycle dependency between the internal structure and the containing class there is always some implicit behavior attached to a class, but an explicit behavior can be used to dynamically control the creation and termination of part instances or to handle communication between the containing class and its parts. Graphically, a small state symbol attached to a port indicates a behavioral port.

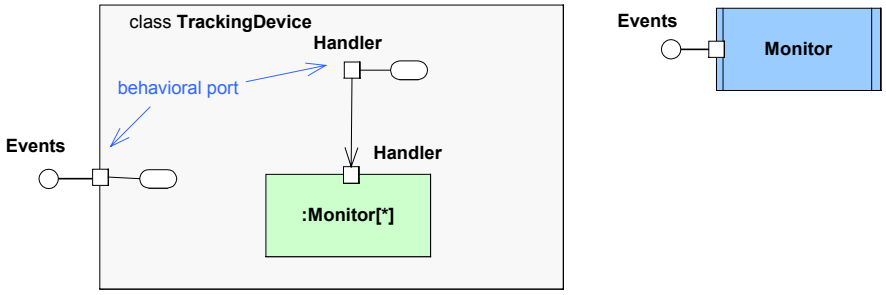


Fig. 7. A behavioral port is connected directly to the behavior of the container class rather than to one of the parts. Alternatively, it allows a part to communicate with the container class, and both cases are shown here. In this example, Handler is a protected, required behavioral port of TrackingDevice, while Events is a public, provided behavioral port.

In UML-RT, a behavioral port corresponds to an *end port*, while other ports are *relay ports*. SDL does not distinguish between gates that connect to the internal structure and gates that connect to the behavior of the agent.

5. Taking Visual Modeling a Step Further

Visual modeling has for a long time been about creating a specification from which application code is more or less automatically derived. With the advent of action semantics for the UML the boundaries between modeling and programming are becoming less clear since it becomes possible to directly execute UML models [Bj01]. This is not a new technique and very much resembles the path taken by SDL, which went from being a pure specification language to more and more often be used as a programming language.

The direct gain of being able to execute models is that it becomes possible to verify system functionality at a much earlier stage in development, and also to automate testing to a large degree. In addition, it opens the door for performance simulation and other analysis techniques that are important when dealing with real-time systems.

The key here is that a part that was underdeveloped in UML 1.4, the available actions and their semantics, have been given a much more precise definition. The action semantics for the UML is currently being incorporated with UML 1.4 in a release called UML 1.4.1, but it is also being integrated with UML 2.0. The main idea is to evolve the code insertion approach previously mentioned, and make sure that UML has the necessary constructs to model detailed behavior precisely. This includes the ability to describe loops, decisions, assignments, calls, and other actions or statements. The abstraction level of programming is at the same time raised quite significantly, because it is possible to generate code that is optimized in different ways depending on the application, and there is no need to explicitly represent pointers, memory allocation, etc.

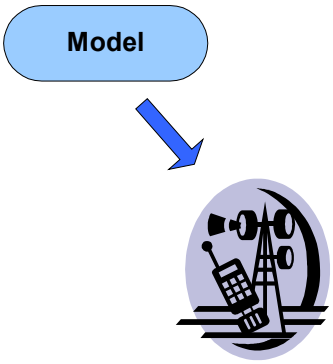


Fig. 8. An executable model can in theory be compiled directly into an application. It is, however, more practical to first transform it to an intermediate format in some programming language. Given that translation rules exists, it is possible to generate code in virtually any programming language; the code is complete and should generally not be touched (cf. generated assembler code or p-code).

In order to fully benefit from executable models, it is required that a model can easily be configured. The primary way this is handled in UML is through the use of the profiles mechanism, which allows a model to be marked up for different purposes. Code generation rules are normally tool specific, but it is possible to define profiles to accomplish this task or to give additional hints to a code generator, for example to indicate whether a component should be generated as a session bean or entity bean in EJB.

The executable model is programming language independent, and depending on which transformations are available in a tool it is relatively easy and straightforward to change from for example Java to C++ as the intermediate format. Additionally, round-trip engineering is not used, since all changes are made directly in the model or by changing the translation rules from UML to the specific programming language.

Furthermore, the model is platform independent, and does not have to capture for example the fact that its different parts should be executing in a distributed environment. All transport mechanisms, encoding, and decoding can be provided by a code generator, and is only dependent on the way a model is deployed.

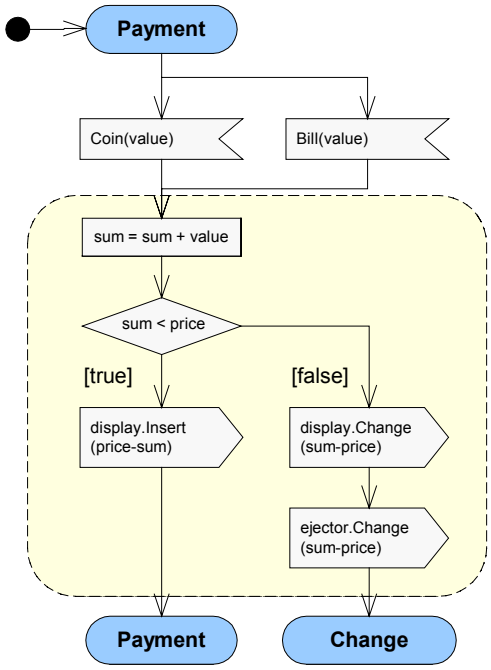


Fig. 9. The action semantics for UML focus on specifying a precise semantics of the parts that are highlighted in this example of a partial state machine. The action semantics does not, however, define a notation to be used for the actions; for this, it is necessary to define a profile on top of the action semantics. Note that in this figure we use a more transition-centric view of a state machine than is customary in UML 1.x. This view is particularly useful when talking about the actions of a transition, and in UML 2.0 complements the traditional state-centric view that could also have been used.

Since a model is executable, it is possible to create an IDE (tool) that is based on UML alone, complete with a debugger that allows you to set breakpoints and watch variables as they change. In addition, it is possible to graphically follow the execution of for example a transition between two states in a trace, and at the same time log the communication between selected parts in a sequence diagram.

6. Schedulability, Performance, and Time

The UML profile for schedulability, performance, and time is often referred to as the Real-Time UML profile. Its focus is primarily on hard real-time aspects, as the main intent is to support modeling of characteristics used to support schedulability and performance analysis. In the schedulability domain, particular emphasis is put on capturing Rate Monotonic Analysis (RMA), as this is the predominant technique

currently supported by tools. However, the standard is generic enough to support other methods.

The profile defines a conceptual model that is based on capturing quality of service (QoS) characteristics, and this conceptual model is then translated into a proper UML profile that primarily captures concepts such as work, period, deadline, worst case execution time, and other properties that need to be supported for the analysis techniques to work.

The entire profile is built around the concept of a resource and someone that uses that resource. The client expects to get some required QoS from the resource, while the resource is actually capable of delivering an offered QoS. In general, there is a problem if the required QoS is greater than the offered QoS.

The general idea is that a UML model is first annotated with information relevant to the problem at hand, for example the execution time of an operation and the amount of time that it blocks. Once the model has been annotated, the information is fed to an analysis tool, for example specializing in schedulability analysis. The results of the analysis may then be fed back to the model to indicate optimal settings given a specific architecture.

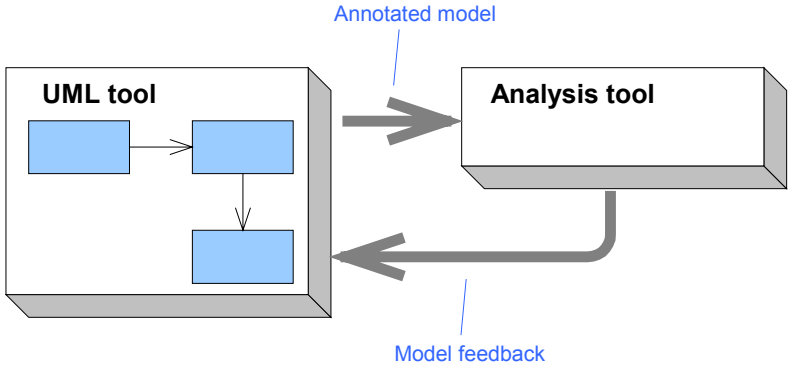


Fig. 10. A model-editing tool is annotated with information required to perform a particular kind of analysis, and then fed to a tool capable of deciphering the information. Based on the analysis, it is possible for the analysis tool to feed back or drive the update of the model according to the outcome of the analysis.

7. Concluding Remarks

Within the Object Management Group, the Real-Time Special Interest Group has made much progress when it comes to looking out for real-time concerns with UML. Just recently the SIG was promoted to a Task Force, which should make the push even stronger. In addition, several of the members are actively working to make sure that UML 2.0 is a suitable language for modeling real-time systems by submitting to the RFPs, and these members also include some of the creators of SDL and UML-RT.

8. References

[Bj01] Björkander, M.: Graphical Programming using UML and SDL, IEEE Computer 24, pp30-35, 2001

[ITU00] ITU-T: Z.100 Specification and Description Language (SDL), 2000

[OMG02a] OMG: Action Semantics for the UML, ptc/02-01-09, OMG, 2002

[OMG00a] OMG: UML 2.0 Diagram Interchange RFP, ad/01-02-39, OMG, 2000

[OMG00b] OMG: UML 2.0 Infrastructure RFP, ad/00-09-01, OMG, 2000

[OMG00c] OMG: UML 2.0 OCL RFP, ad/00-09-03, OMG, 2000

[OMG00d] OMG: UML 2.0 Superstructure RFP, ad/00-09-02, OMG, 2000

[OMG02b] OMG: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms RFP, ad/02-01-07, OMG, 2002

[OMG02c] OMG, UML Profile for Schedulability, Performance, and Time Specification, ptc/02-03-02, 2002

[OMG01] OMG, Unified Modeling Language Specification, version 1.4, formal/01-09-67, 2001

[SR98] Selic, B., Rumbaugh,. J.: Using UML for Modeling Complex Real-Time Systems, Industrial white paper, Rational, 1998

[U2P02a] U2 Partners: UML: Infrastructure version 2 beta R2, ad/02-06-01, 2002

[U2P02b] U2 Partners: UML: UML 2.0 Proposal version 0.671, http://www.u2-partners.org, 2002

[UU02] Unambiguous UML: Submission to UML 2 Infrastructure Submission, ad/02-06-07, 2002

UML for Embedded Real-time Systems and the UML-Extensions by ARTiSAN Software Tools

Andreas Korff

ARTiSAN Software Tools GmbH
Eupener Straße 135-137
D-50933 Köln
Andreas.Korff@artisansw.com

Abstract: Modelling software using the Unified Modelling Language (UML) also for embedded real-time systems (ERS) becomes more and more popular since the complexity of these systems increases as well as the pressure of short time-to-market timescales. These notes will introduce some extensions to the UML notation implemented in the CASE-tool Real-time Studio, their motivation and their integration with the common UML modelling procedures to ensure a complete picture of the embedded system to be developed.

1 Introduction

The UML specifies in its current version 1.4 its intent to visualize, specify, construct and document the artefacts of software-intense systems. The UML is not intended to be a visual programming language. Instead, it is named as „visual modelling language“. So the UML is directly focussed on solving the problems occurring during the development of software – also within ERS:

1. A common understanding of the system requirements for all members of the development team (including the project sponsors). This helps to avoid the biggest thread in developing systems or software: The solution or product does not fit to the (real) requirements.
2. The possibility to present the system (requirements and solutions) definitely and from different perspectives and in different abstraction layers overcomes the possible misunderstandings when using only textual descriptions.
3. The re-use of solution strategies through object orientation: The goal is to develop system components with a well-defined interface to the external world. Instead of decomposing functions in order to handle high complexity we model the responsibilities of subsystems or objects. These are therefore easy to extend or to replace without disturbing the rest of the system.

This points are valid for all software projects. During the development of ERS, there is another problem: The different worlds of systems engineers, hardware and software engineers have to be combined and bridged in a linguistic and conceptual way. Systems

engineers are thinking in abstract solutions, hardware engineers are using integrated circuits and hardware components, and software engineers always express their ideas using algorithms and data or object structures. The quality of an ERS depends on the ability to integrate these "worlds" of thinking. The UML itself heavily is software-minded, so it is rather difficult to express solutions or ideas independent from software or to describe the system's hardware topology in a detailed way. Additionally, the real-time aspects within the software itself are also difficult to express in plain UML. But it is possible to fill the notational gaps for the modelling of ERS. By doing this, it is important that the UML extensions fit to the UML notation and are usable and readable intuitively.

State-of the-art system (sub-)functions are very complex. But also their interlocking makes it very complicated to project and to implement this functions. The UML approach is helpful here by encapsulating functionality as use cases. The uses cases afterwards can be implemented step by step.

2. An Example System

ARTiSAN offers prospects the opportunity to evaluate the CASE-tool Real-time Studio Professional for a limited time. Within this evaluation, it is useful to show the tool handling , the different diagrams and what is possible to do with a model by some small examples. These examples, however, still are real-time and embedded. One of this examples, a well-known parking lot, is used here to show the typical development of an ERS.

2.1 The Requirements Analysis

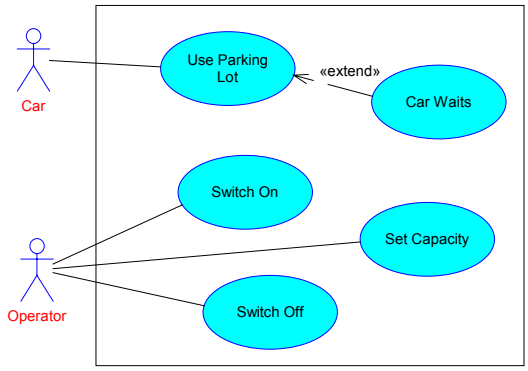


Figure 2.1-1 The Use Case Diagram

Before implementing a system, it is necessary to examine the given requirements in order to avoid gaps or inconsistencies. Within the UML, it is possible to transform functional requirements directly to use cases. In our parking lot example, the identified actors "Car" and „Operator" are related to the system functionality they are using. The use case diagram in Figure 2.1-1 shows the appropriate relations.

The use case description gives the possibility for the system engineers (and the customer) to describe in a textual way, how the interactions are working between outside (i.e. the actors) and inside the system as well as possible pre- and post-conditions.

Another important aspect of ERS of course refers to the real-time behaviour. In the beginning of requirements modelling they can be expressed as optional property of use cases. Figure 2.1-2 shows this timing note tab for use cases. Because timing is defined as non-functional requirement, this type of information can be collected in the area of non-functional constraints.

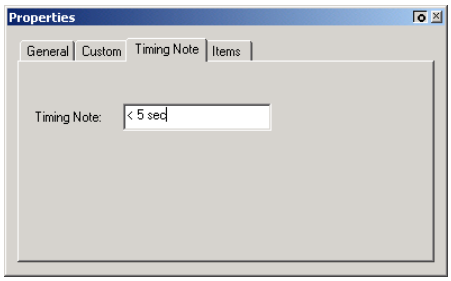


Figure 2.1-1 Use Case Timing Note

Unfortunately the UML does not contain a precise possibility to describe the interfaces at the system boundary in a sufficient way for ERS. Typical for them, the interaction with the outside world does not only happen with „normal" actors, but also other types of actors like different systems, sensors or generic entities like "the time". Therefore it is necessary to have a very detailed, but non-software-driven definition of the exact system boundary and of the interfaces the actors can use to interact with the system. In order to support this, ARTiSAN has introduced the context diagram as an UML-Extension within Real-time Studio. In this sub-type of a system architecture diagram, it is possible to show the interface devices related to the specific actors as well as subsystems within the ERS in order to divide the system into functional sub-entities, if necessary. The control system (or software) is drawn as a separate subsystem, thus as black box, connected to the interface devices or other subsystems using communication links for incoming or outgoing messages.

In the parking lot example, we use interface devices to model the sensors realizing that a car wants to enter or to exit the parking lot. Additional devices like push buttons, key pads or displays define the access an operator has to interact with the control system, e.g. to change the number of allowed cars inside the parking lot. Using this type of modelling technique, the use cases for these actors can be modelled in a much more detailed way. Figure 2.1-3 shows the context diagram for the parking lot. The events added onto the information links between the different model elements are used throughout the whole model and their contents are added when available. As an example, there is a "break" event coming from the infrared beam sensor signalling the control system that a car wants to enter the parking lot. This "break" can and will be used in object sequence

diagrams for use case descriptions, in other system architecture diagrams for topological information or in a system state diagram to define the reaction of the system as a whole.

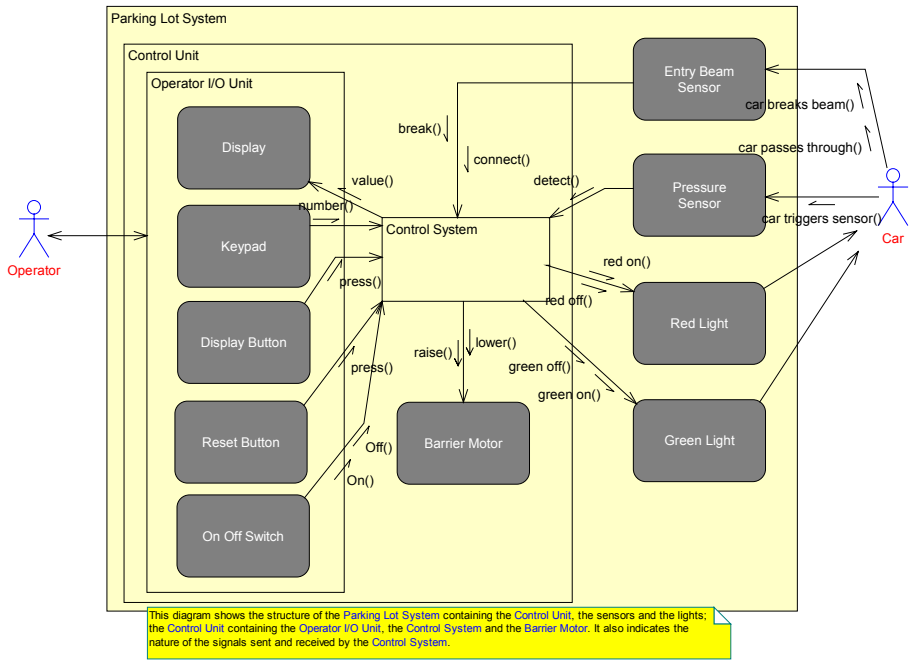


Figure 2.1-3 The Context Diagram

A specific fact that qualifies ERS is the impact of non-functional requirements on the system architecture and the system design. If there are functional limitations, they can be repaired or improved most of the times by additional software updates. Errors or limitations in the area of non-functional requirements like e.g. reliability, timing, size or costs often result in a completely erroneous system architecture, thus resulting in the complete project to fail. In order to repair them, the whole development project must be restarted from the beginning, if possible. Within Real-time Studio, the importance and therefore the necessity to include non-functional constraints into an UML model is covered by the UML extension of Constraints Diagrams. As the name “constraints” implies, this type of requirements have to be linked to the related functional requirements (i.e. use cases), so their range of valid implementations is constrained by the relevant non-functional requirements. In our example, the non-functional constraint “Barrier Motor” defines the time maximum which is allowed for opening or closing the barrier. The use case “Use Parking Lot” is related to this constraint, therefore its design and implementation must follow this timing. Figure 2.1-4 shows the appropriate constraints diagram together with the links editor where constraint and use case can be linked.

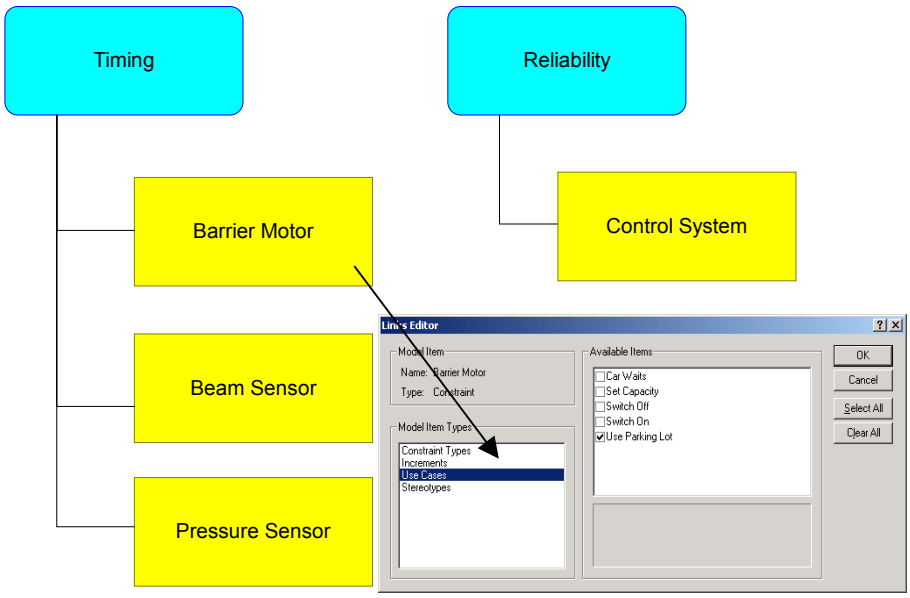


Figure 2.1-4 The Constraints Diagram and possible links to Use Cases

For the requirements analysis of the parking lot model, it is also necessary to formalize the more complex use cases like „Use Parking Lot“. Inside the use case descriptions, the only way to define the possible scenarios is by textual means. An object sequence diagram is a more formal way to describe the use case scenarios, especially taking into account the ARTiSAN extensions for this diagram: Each sequence step is shown in this diagram not only using the object interaction, but also by a textual description. So the transition between the pure textual use case description and a diagram is much easier. Also additional sequence structures can be used inside this textual descriptions like selections, iterations or parallel sequences. This helps to restrict the number of scenarios (or object sequence diagrams) necessary to fully describe the use case. The object sequence diagram itself contains rather more abstract objects like actors, interface devices, subsystems or event. The software control system, which contains the software objects we currently don’t know at this stage, is modelled as black box similar to the context diagram. Two additional graphical items are worth mentioning, too: The system boundary, which was defined already in the context diagram, divides as think line the external actors from the internal system elements like interface devices or software objects. These two types can be differentiated using a dashed line named architectural boundary. This object sequence diagram usage for requirements analysis is shown in Figure 2.1-5.

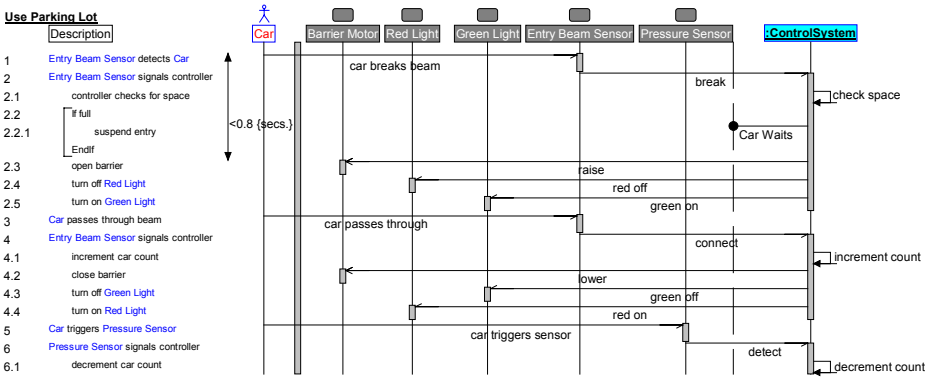


Figure 2.1-5 The Object Sequence Diagram formalizing an Use Case

As a result, the use cases form the basis of the requirements analysis. But to describe the use cases as exact as possible also in their relationships need notational support beyond the "extends" or "includes" typical for UML. For instance is can be necessary to define that the start of a use case is not always possible, some use case scenarios might be mutually exclusive or they might depend on each other. In order to model the overall system behaviour to external events in the granularity level of use cases, a system state diagram can be drawn. The same graphical notation as for state diagrams showing dynamic object behaviour can be used within this diagram. The transition triggers, of course, are the same events running through the system boundary in the context diagram, and the actions within the event action blocks are modelled on use case level.

2.2 The Solution Architecture

In order to create a design model for system implementation, the UML offers a rich set of notations for the object design. Object interaction can be expressed in object collaboration diagrams or object sequence diagrams, for the static object design class diagrams can be used and the dynamic behaviour of objects are modelled in state diagrams. All these features are defined in the UML specification and are located in the object architecture, one of the three abstraction layers in the ARTiSAN incremental, iterative development process "The Real-time Perspective".

The other two, the system architecture and the software architecture, are specific for ERS, and are described below.

Topological Information about where objects are instantiated in the system, can be expressed in the UML using components diagrams or deployment diagrams. These diagrams cannot reach the depth of detail necessary for ERS. In order to integrate new hardware and software smoothly, additional information like memory-mapping, hardware port addresses, processors and processor types, boards and board types containing board I/O devices, etc. is very helpful.

The interface devices defined in the requirements architecture can now be linked to their appropriate board I/O device with its software specific interface, e.g. a port or register. With this information, the software developer knows the direction to work to very precisely. The common UML model therefore forms the clear communication basis for the development team. Ambiguous ideas, concepts or views, or gaps in the proper definitions can be identified in an early stage and can be solved within the team. For the parking lot example, figure 2.2-1 shows as system architecture diagram the hardware details of the control system:

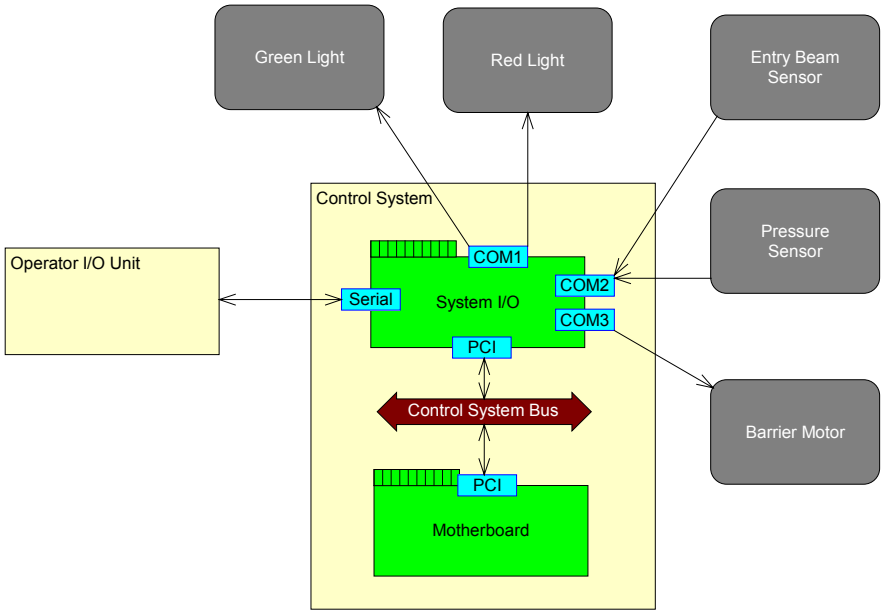


Figure 2.2-1 The System Architecture Diagram

Also on the hardware side, re-use shall be one of the most important design goals. Targeting this, all relevant information must be described using the appropriate aspect. So the same type-instance-model as for the software is used. If e.g. a specific VMEbus board is used in an application, its start address is a property of this one instance of the board. On the other side, the board I/O devices belong to the board type and thus are stored within the board type properties.

For the development of an optimal solution for the real-time aspects of an ERS given in the requirements analysis phase, it is necessary to build up a conception of the relevant concurrent thread or tasks and their communications flow. This concept can be created stand-alone without being mixed with other items of the software object design in order to develop a clear picture of the software architecture. Within Real-time Studio, the UML extension of the concurrency diagram enables the software designer to express his concurrency concepts. Figure 2.2-2 shows the ideas for the parking lot example. From the requirements model, it is clear that both the control of cars entering or exiting the

parking lot and the operator's access to manipulate the number of allowed cars should run in parallel. In order to support fast reaction times for the complete system, a third task is modelled for event detection. This task is connected to the input interface devices like the sensors, the keypad or the discrete pushbuttons. If an external event occurs, the event detection tasks informs the consuming tasks via message queues, event flags, mail boxes or other inter task communication means. These and also the tasks can be linked to the class model within the model item properties. Thus a direct navigation to the relevant class model elements implementing the given concurrency concept is possible.

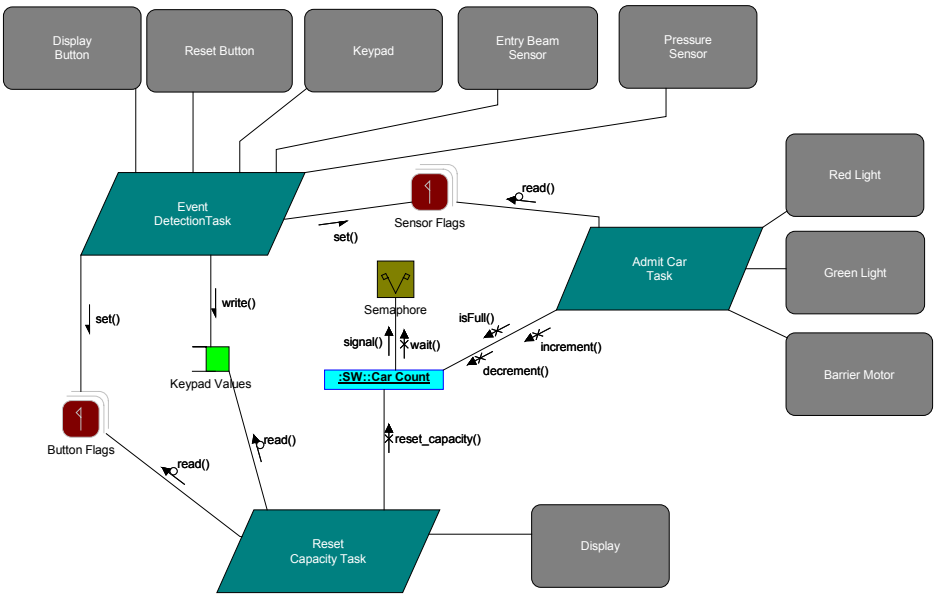


Figure 2.2-2 The Concurrency Diagram

Considering the task sequences themselves, their timing has to be checked against the timing requirements for the events stimulating the task. For this objective, an object sequence diagram can be used as a tasking sequence diagram. All timing information, constraints, timing budgets, calculations and measures are collected here for a given task, thus forming a complete picture by which it is possible to analyse if the requirements can be met. This analysis is supported by the fact, that already every event or message can contain response duration and detection lag timing. The analysis about timing and schedulability can be made either manually or automated by tools. The forthcoming UML Real-time Profile –now in finalization phase in the OMG's Real-time Analysis and Design Working Group will provide a standard set of stereotypes and tagged values. So the modeller can properly define the timing and schedulability constraints in the model, whereas an analysis tool using e.g. the rate-monotonic analysis method can access these data and calculate the corresponding results.

3. Summary

Only with the ability to model information, requirements and solution ideas specific to embedded real-time systems, a complete picture is created for all people involved in the development. Using this, a UML model is created to support the goals of the development project:

- An unambiguous communication basis for all the members of the development team
- A fitting notation for all information layers and –types within the requirements analysis and the solution architecture

All information is stored consistently in a model database. During the development project, the model information is condensed, so an object-oriented solution and implementation can be made up, perfectly matching the given requirements. This is supported by the incremental and iterative development process, which is loosely integrated into the modelling tool. The process can be used as overall development procedure or as on-line help, showing the “red line” through system and software development. If necessary, the process can be adapted according to the relevant company regulations.

In order to use the modelling information, several add-ins are provided. Code synchronizer generate C, C++, Java or Ada Code from class model information, synchronize the differences between code and model level, or reverse-in existing code into UML class model information. The document generator is able to generate according given and adaptable templates the documents fitting the needs of the appropriate project phase. The templates provided out of the box are directly related to the project documents defined in the development process. They can easily be adapted to fulfil the documentation style regulations as well as in the way information is gathered from the model into the documents. There is a model merging tool, and also tools for generating SQL or CORBA IDL statements in order to support distributed systems. Simulation of object interaction up to the simulation of dynamic object behaviour and its link to a graphical prototyping tool complete the tool support to develop the right ERS product the first time.

Bibliography

[OMG02] OMG web site URL: http://www.omg.org/

[Art02] ARTiSAN, Real time Studio Professional®
http://www.artisansw.com/pdflibrary/rtspro_ds_4.pdf

[Art01] ARTiSAN, Real-time studio, Installation Guide, Version 4.1

[MM98] McLaughlin M.J, Moore A., Real-Time Extensions to UML, Dr. Dobb's Journal, December 1998.
http://www.ddj.com/documents/s=913/ddj9812g/9812g.htm

[MC98] Moore A., Cooling N., Real-Time Perspective, Foundation.

http://www.artisansw.com/whitepapers/rtsfoundation.pdf

[MC98_1] Moore A., Cooling N., Real-Time Perspective, Overview

http://www.artisansw.com/whitepapers/rtsoverview.pdf

[Art98] Real-time Perspective Mentor: http://www.artisansw.com/mentor/start.htm

[Art00] ARTiSAN, Model management in Real-time Studio, 2000

[Art01_1] ARTiSAN, Real-time studio®, User's Guide, Version 4.1

OMER-2 Workshop
Daimler-Chrysler Modeling Contest

Modeling S-Class Car Seat Control with AnyLogic

Alexei Filippov alf@xjtek.com,
Dr. Andrei Borshchev andrei@xjtek.com

St. Petersburg State Technical University,
XJ Technologies
http://www.xjtek.com
Fax: +7 (812) 2471639
21 Polytechnicheskaya street
St. Petersburg
Russia

Abstract: In this paper we give an overview of the car seat model that was developed for Daimler-Chrysler modeling contest in year 2001 and was awarded the 1st prize. We outline the OO UML-RT based modeling approach that was used and the simulation tool AnyLogic that supports it, and discuss their main advantages with respect to automotive area.

1 Introduction

Many different object-oriented methods for the development of embedded real-time systems have been made public during the last years. Some of them are already supported by off-the-shelf tools, others are still research prototypes. From the viewpoint of industry research, Daimler-Chrysler decided to identify the best development method. The comparison was performed in the form of a contest, where different methods were applied to the same problem. The real-life working specifications of the S-class car seat control were offered as the problem definition.

With over a decade experience in OO modeling and real-time systems, St. Petersburg Technical University and XJ Technologies took part in the contest. The model of the car seat was developed with the modeling and simulation tool AnyLogic from XJ Technologies that supports the extended UML-RT as the modeling language.

The result of that development work – an executable animated model of the car seat controller, conforming all specifications and supporting the predefined interfaces – demonstrated that the modeling approach supported by AnyLogic is highly applicable to the automotive area. The model appeared to be very compact, precise, well-structured and intuitive; it was awarded the 1st prize at the contest.

2 The Modeling Language of AnyLogic

The Unified Modeling Language (UML) that originally was proposed to handle complexity in software systems, has proven to have a strong set of concepts applicable across domains. AnyLogic utilizes the power of UML-RT (UML for Real Time, an extension of UML that roots to ROOM language) in simulation applications. AnyLogic supports those constructs of UML-RT that are necessary to construct fully executable models of high expressive power in multiple application areas. AnyLogic extends UML-RT so that the resulting language is sufficient to construct such models. Detailed information about the modeling language can be found in [BKR97], [BKS00] and [Bo01].

When developing a model in AnyLogic you develop classes of "active objects" representing real objects that have activities and states inside and interact with their surroundings. Active objects interact solely through interface elements - ports and variables. Active objects may inherit properties and encapsulate each other to any desired depth, so that the model is structured as a hierarchy (a tree) of instances. Behavior specification is strictly separated from structure.

Behavior in AnyLogic is specified in terms of statecharts, timers, events, and ports. Statecharts (enhanced UML state machines) are used to characterize event and time ordering of operations. They specify the states of the active object and transitions between them. Messages are used to model various information units passed between active objects. They also can inherit properties and encapsulate other messages. Messages are sent, received, and routed at ports. One can define arbitrary queuing and routing policies.

Interface variables are a useful extension to UML-RT supported by AnyLogic. They may be exposed at the active object interface and connected to other objects. One can define algebraic and differential equations over entities to model e.g. the physics of the evironmnet the control system is embedded. Moreover, one can embed sets of equations into the statechart states to capture compex interactions of discrete logic and continuous behavior. This enables hybrid simulation – the feature highly demanded by embedded system developers.

3 The Car Seat Model

The car seat has two groups of adjustments, three adjustments in each, memory for two seat positions, courtesy seat adjustment for climbing in and out, and two heating stages. There is a number of implementation restrictions, for example only one of two motors can be activated at a time, heating should be turned off when the motor is on, the motor should be turned off before reaching the end of movement range, voltage conditions, etc.

One of the primary contest requirements to models was a clear and natural division of the model into components. The model of the car seat is partitioned into several active object classes according to the original functional specification [DC00]. The structure of the main model class Controller is shown in the Figure 1.

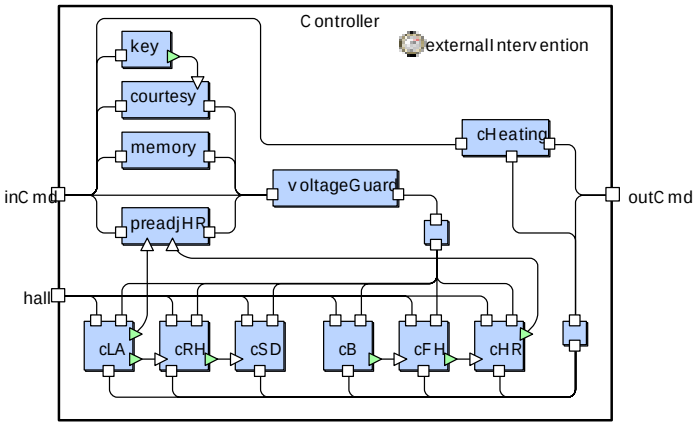


Figure 1 Structure of the Controller Class

The Controller object is composed of several sub-controllers:

- Ignition key sub-controller handles ignition key operations, enables and disables controller circuits according to the key state;
- Memory sub-controller handles position memory functions, allowing storing and restoring car seat positions;
- Courtesy adjustment sub-controller provides automatic movement of the seat to the backward position to help driver climb in and out of the car;
- Head restrain pre-adjustment sub-controller automatically adjust head restrain depending on the seat position;
- Heating sub-controller handles heating device functionality, switching between heating modes 1 and 2; temporarily turns heating off during seat movement operations to reduce power consumption.
- Motor sub-controllers. These six sub-controllers correspond to six possible movements of the seat. They also handle motor priorities.

The other essential feature of embedded and real-time systems in general, and the car seat in particular, is event- and time-ordering of operations. To efficiently specify that type of behavior we have intensively used the statechart notation. It is very powerful and flexible for describing various classes of algorithms, especially for reactive systems, when reaction on an event depends on the current system state. As a statechart example we could consider the memory controller algorithm show in the Figure 2.

The functionality implemented by this statechart is the following. The user has three buttons M, M1, and M2. It is possible to store up to two seat positions in the car memory. When the user presses button M and then within 2 sec either button M1 or button M2, the system stores the current seat position to the corresponding memory slot. If no button is pressed within 2 sec time interval after button M, the storing process is aborted. On the other hand the seat position restoring process is initiated when the user presses

either M1 or M2 buttons. Then the seat starts moving to the stored position. The movement stops when the final position is reached or if the user releases the button.

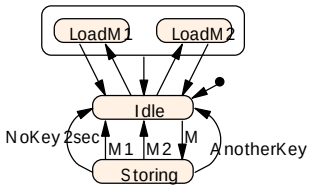


Figure 2 Memory Controller Main Statechart

Another functionality structuring technique that was used in the model is command pipelining. The user commands go though several processors and filters before they reach the main controller as shown in Figure 3. This pipelined architecture naturally lies on the UML-RT paradigm of messages, ports and statecharts.

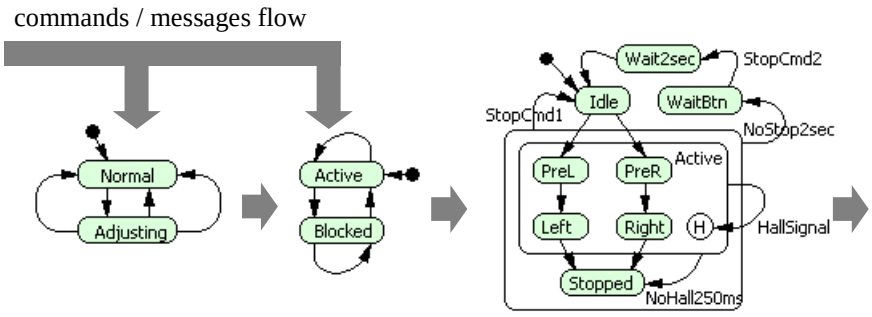


Figure 3 Command pipelining

An interactive animation was developed to enable visual testing, debugging and demonstration of the executable model. The animation displays the seat position, the motor and heating states, and has a number of controls via which the user can issue adjustment commands, use the position memory buttons, turn the ignition key, model the car speed, door opening, and voltage conditions. As AnyLogic generates 100% Java models, the animated model can be run over the Internet in a web browser. The model is available from XJ Technologies site at http://www.anylogic.com/applications/?model=carseat.

4 Conclusion

The main result of the modeling work is the proof of applicability of AnyLogic / UML-RT modeling language to the automotive area. The model developed is very compact and concise; its structure naturally reflects the problem logic. The two main diagram notations of UML-RT were extensively used: the structure/collaboration diagrams and statechart diagrams. Ports and message passing mechanism were used for command pipelining, and statecharts – for definition of the controller logic, including timing and ordering of operations. Inheritance enabled implantation of the functionality common to e.g. several subcontrollers in a base subcontroller class. Moreover, the ability of AnyLogic to model continuous and hybrid processes made it possible to put the controller into the experimental environment within using just one tool on one computer. In general, the approach that we used allowed us to spend very little time on adjusting the tool to our needs and to concentrate more on the essence of the problem.

Bibliography

[BKR97] A. Borshchev, Yu. Karpov, V. Roudakov. Systems modeling, simulation and analysis using COVERS active objects. Proceedings of the 1997 Workshop on Engineering of Computer-Based Systems (ECBS '97) Monterey, CA, March 1997.

[BKS00] A. Borshchev, Yu. Kolesov, Yu. Senichenkov. Java Engine for UML Based Hybrid State Machines. Proceedings of the 2000 Winter Simulation Conference, WSC'2000, Orlando, FL, December 2000.

[Bo01] A. Borshchev. AnyLogic 4.0: Simulating Hybrid Systems with Extended UML-RT. To appear in: Simulation News Europe, Issue 31, April 2001.

[DC00] Daimler-Chrysler, Detailed Functional Specification of the Car Seat Model, http://www.automotive-uml.de/mc/index.html

[XJ00] Experimental Object Technologies. AnyLogic 4.0 User's Manual. Can be downloaded from http://www.xjtek.com/products/anylogic within the preview version of the tool itself.

Development of a Car Seat: A Case Study using AUTOFOCUS, DOORS, and the Validas Validator

Peter Braun
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
D-85748 Garching b. München

Oscar Slotosch
Validas Model Validation AG
Software-Campus
Hanauerstr. 14b
D-80992 München

Abstract: In this paper we describe the modeling process and the resulting model of a typical car seat. The requirements of this seat are documented in [Chr00] which are the input of our process. We used the tools AUTOFOCUS [AF-02], DOORS [Tel02], and Validas Validator [Val02]. Starting with requirements analysis we develop first model fragments. Afterwards the graphical, component oriented approach of AUTOFOCUS is used to model the system. Requirements management and tracing techniques ensure that all requirements are implemented. The model-based core of the development process helps very much for the requirements tracing. The model fragments of the earlier phases can be updated so that tracing information is consistent. Compared to traditional requirements tracing techniques less manual interaction is needed.

Beside this the test management is also done based upon the requirements. For relevant requirements test cases are specified. This is done using the AUTOFOCUS notation of Extended Event Traces (EETs) a variant of Message Sequence Charts (MSCs). Afterwards the generated code of the model is tested based upon those test cases. Further validation techniques like simulation, consistency, and determinism checks of the Validas Validator have led to the detection of inconsistencies in the model and in the specification.

1 Introduction

In the following an adapted process for the development of embedded systems with AUTOFOCUS, DOORS, and the Validas Validator is shown by an example system dealing with the control of a typical car seat. The process starts with some requirements analysis activities. Our graphical, component oriented approach is used to model the system. Requirements management and tracing ensures that all requirements are implemented.

Component orientation is a special form of object orientation. The used description techniques are similar to UML-RT. We use the tools AUTOFOCUS and the Validas Validator to check consistency and to generate code. One difference to UML-RT is that we use EETs [HMS+98, HSS96] instead of MSCs. EETs have a precise semantics, and can be used to express repetition for a certain time, and to generate tests. Simulation is used for building

a prototype of the software parts of the car seat controller. We implemented a GUI for testing the model, based on the given interface specification.

Components have (compared to objects and classes) some advantages. They are static, so their size can be determined which is important for efficient C code generation. The simple, well typed communication model used allows to determine the size for messages and buffers as well. Together with the scheduling concept for the components the model is a perfect base for real-time applications. For prototyping the model we implemented a timer component, and used the Java method `currentTimeMillis()` to access the system time.

In the development of the model we put more effort to the process (requirements structuring), modeling, and prototyping. C-Code generation was not in the focus, because it was easier to test the Java code and integrate it with a simple GUI. The total amount of time spent with this case study was approximately four weeks.

We start with a short description of the method and refer the applied tools. Section 3 contains a sketch of the model and the applied design principles. Section 4 shows some of the validation methods used.

2 Method

Nowadays even embedded systems become more and more complex. Beside this challenge even embedded systems are heavily interconnected. To guide developers new methods have to be established. Those methods should be compositional and hierarchical so that the functionality of the target systems can be split into different hierarchical components. The methods should support the developers in every stage of the development from requirements engineering over design till validation and test of the developed system. Equally important is that tools support this method.

The UML [UML99] provides notations for the development of object-oriented systems. These notations are loosely coupled and are heavily influenced by object-oriented programming languages like Java, Smalltalk, or C++. The UML defines no method how this language may be used. There are many different methods which use the UML in the context of internet or business applications, but especially in the context of embedded systems there are only first steps towards methods. As the object-oriented approach used in the UML doesn't seem to support the development of embedded systems very well, another component-oriented language named UML-RT will be integrated in future versions of the UML. The notations of this language support hierarchical components, beside some other concepts and fit the needs better than pure UML.

In the following we will describe some facets of our method, which we have used to develop the software part of the seat controller described in [Chr00]. Our method provides a component based approach to develop and describe the software part of embedded systems. We use fewer notations than provided by UML or UML-RT, but most of the notations are very similar to their counterpart in UML/UML RT. A main difference is that our nota-

tions are founded by a mathematical theory and therefore they are integrated very tightly. Note that a developer has not to know this mathematical theory to access the benefits.

As stated above tool support is essential. Our method is based upon a tightly integrated tool chain. The tools we use are DOORS from Telelogic for Requirements Engineering and Requirements Management, our own component-oriented CASE-Tool AUTOFOCUS for the specification of software systems and code generation, and the Validas Validator for validation and verification support.

2.1 Requirements Engineering and Requirements Management

Usually the software development processes starts with analysis and specification of the problem space. The process starts with roughly structured User Requirements which are transformed into System Requirements. This is a very universal process which deals with lots of informal notations and very few structure. Tools like DOORS support those steps by providing the possibility to structure text to some extend and by focusing on requirements management and tracing. The support given by DOORS mainly helps developers to not lose the overview and to manage the relationship between different kinds of information at this early stage. Starting with the System Requirements the step from the problem space towards the solution space has to be done. This step from "What" to "How" or from analysis towards design has to be taken carefully. It is essential that this step is at least traceable. Preferable this step is provided by a continuous method, so that as many links as possible between design and analysis information are generated automatically.

In our method we first import and split the requirements so that they can be managed with DOORS. The result of this step are one or more documents with all System Requirements. The step to these System Requirements from the User Requirements can be supported by DOORS. The resulting documents are plain text, which is structured into smaller quantities which contain requirements. Now these text blocks are classified by their kind. They may specify requirements for the software, the hardware, the environment, or any combination of them. This classification is important as we only concentrate on software. Design of hardware could be carried out in parallel. The decision which parts of the overall system are realized in software and which are realized by hardware can be revised later. The decision that a part is realized in software is a decision, that description techniques provided by AUTOFOCUS are used to describe the functionality of that part.

After this classification a new document containing the requirements for the software system could be generated. The generation ensures the traceability between the original document and the document with the software requirements. Depending on the structure of the original document some work has to be done, to adapt the new document so that it is at least readable and structured appropriately.

The next step is the identification of some coarse model-frame for the design. Therefore the software requirements document is used. Here some requirements are attributed with e.g. components or component-types fulfilling those requirements. Within a new document "model-frame" these recognized component-types are further described. Instances

of component-types can be connected to other component-types using links. The model-frame document contains further model-types and relations.

Using the information in the “model-frame” a so called surrogate module can be generated which is another DOORS document. This document is used to generate a first AUTOFOCUS model, which can be developed further using AUTOFOCUS. As every “object” in this surrogate module has links to the “model-frame” and the software-requirements document and that contains links to the original requirements document it is initially ensured, that a developer can see all requirements which resulted in model-elements. As the model is developed further a translation back to DOORS is possible, using the abovementioned surrogate module (which relates the unique identifiers of model-elements used in DOORS and in AUTOFOCUS). Surely new relevant model-elements must be related to their requirements appropriately.

With this technique it is at least possible to control if all recognized requirements are “satisfied” by some model-element. Further it is possible to show at all requirements for a given model-element, and even more important to identify all model-elements which are related to a requirement. This traceability is very important if some requirements are changed.

Figure 1 shows an overview of the above described process. The process starts with the system requirements. The system requirements are structured and classified. A resulting document containing the software requirements is generated. This document has to be further refined and structured. By identifying some model-elements within the software requirements and by providing a model-frame, some first fragments of a model can be generated. In parallel the development of test-sequences ensuring some software requirements can be described within the test-frame. From the test-frame EETs containing these test-sequences can be generated.

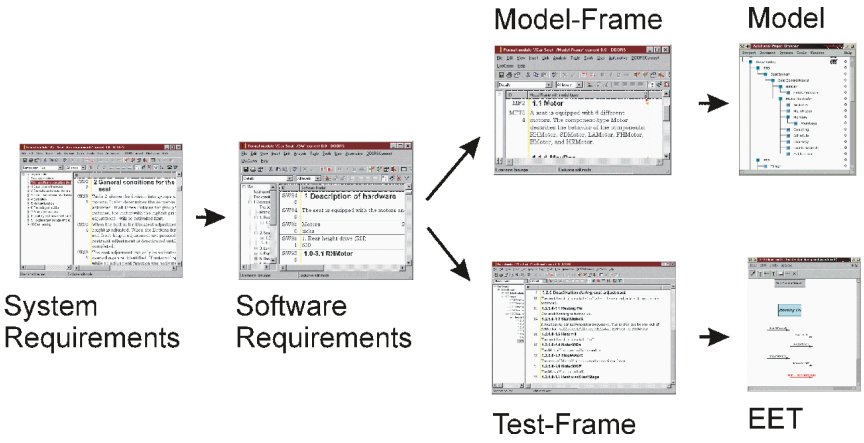


Figure 1: The overall process

2.2 Design

The design phase the model build upon the recognized model-elements is further developed. The design is carried out using AUTOFOCUS which supports an important subset of UML-RT:

- System Structure Diagrams (SSDs) describe the system structure and the interfaces of the components.
- State Transition Diagrams (STDs) describe the behavior of the System.
- Data Type Definitions (DTDs) define data types which are e.g. used for the specification of communication channels of the system. Together with the data types auxiliary functions dealing with them can be defined.
- Extended Event Traces (EETs) describe the dynamic behavior by example communication sequences between the components. EETs are automatically generated during simulation or from other validation techniques.

All description techniques are hierarchical, so that the model can be structured appropriately.

The system is developed top-down, i.e. the structure is designed hierarchically. Additional requirements lead to refinements or incremental changes of the system. It is easy to extend the interfaces in order to send additional messages required for additional features. Requirements tracing allows the designer to check if all features have been modeled and tested.

2.3 Validation

Building a model with AUTOFOCUS is quite simple, however building a correct model is not. The first step is to check the consistency of the model. This allows to detect for example unconnected channels or misspelled ports in transition diagrams.

The next validation step is to simulate the model (or parts of it). Simulation shows the developer the dynamic behavior of the system and allows to debug the behavioral descriptions. Within the simulation the model is animated according to given inputs using the input examples of the user.

Additional validation techniques allow the developer to detect nondeterminism in the models and to verify that no messages are lost, or certain inputs lead to certain outputs. In the example all components are simple enough to apply formal methods (model checking) for validation, however the whole system cannot be model checked.

Simulation cannot guarantee correct models as usually only some trace through the system can be tested. Unfortunately automated validation techniques like model checking also cannot be used for most practical relevant systems as those systems are often already to

complex. But model checking can be used to ensure the correctness of some smaller parts of the system. Often even those smaller parts have to be modeled more abstract with e.g. restricted data types. So this is usually only done with some critical parts of the system.

After the validation of the components and the system the integration test is build. The generated code supports textual inputs (test driver generation). This allows the automated run of test cases specified by text files.

In addition to the integration test we build a GUI for testing the system using the given interfaces.

3 Model

In this section we describe important principles of the process and the model.

3.1 Requirements Engineering and Requirements Management

As described in Section 2.1 we start with the requirements specification of [Chr00]. First we imported the original text into DOORS. Thereby we divided the text into smaller text blocks (requirements). After that those requirements are classified as described above. In Figure 2 some parts of the specification describing the seat heating are shown.

After the classification into requirements for the hardware, the software and the environment a document containing the software requirements is generated. This document is very similar to the original document as the original document already is focused on the software. This is not the general case as some case-studies with BMW have shown.

During further "development" two main components in the "SeatControlModel" are identified. The Heater deals with requirements of the seat heating, the MotorController handles the control of the motors. Within the MotorController, components dealing with the memory, the switches, and the hall sensors are identified. Further components for the core control of each motor are identified.

The document shown in Figure 3 contains these initial model elements. It describes components and component-types. Some further attributes of those components are identified and described here too. E.g. local variables of each motor containing their current position and their maximum positions are identified in this phase.

Starting with these components further modeling in AUTOFOCUS is carried out. To control the development at some stages where the model or parts of it seems to have reached a stable state, the back-translation to DOORS was done. Here the description of the model-elements had to be further refined and the links of the model-elements have to be adapted. So after these steps one can see if there are further requirements, which are not yet "satisfied" by a model-element. The relation between model-elements in AUTOFOCUS and DOORS is stored in a surrogate module. This surrogate module is normally hidden.

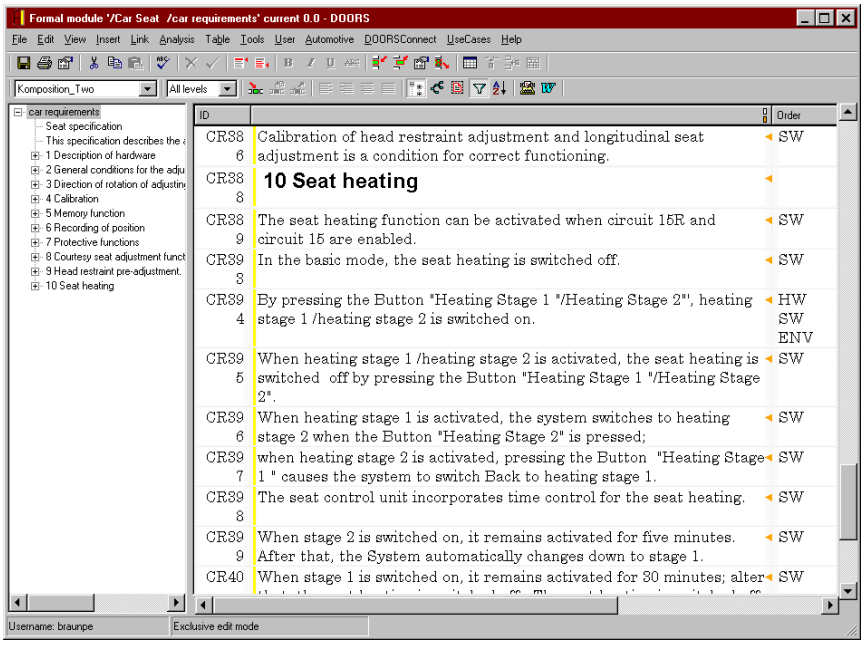

Formal module '/Car Seat /car requirements' current 0.0 - DOORS
File Edit View Insert Link Analysis Table Tools User Automotive DOORSConnect UseCases Help
Komposition_Two
All levels
car requirements
Seat specification
This specification describes the
1 Description of hardware
2 General conditions for the adju
3 Direction of rotation of adjustin
4 Calibration
5 Memory function
6 Recording of position
7 Protective functions
8 Courtesy seat adjustment funct
9 Head restraint pre-adjustment.
10 Seat heating
ID
Order
CR38 6 Calibration of head restraint adjustment and longitudinal seat adjustment is a condition for correct functioning. SW
CR38 8 10 Seat heating
CR38 9 The seat heating function can be activated when circuit 15R and circuit 15 are enabled. SW
CR39 3 In the basic mode, the seat heating is switched off. SW
CR39 4 By pressing the Button "Heating Stage 1 "/Heating Stage 2"", heating stage 1 /heating stage 2 is switched on. HW SW ENV
CR39 5 When heating stage 1 /heating stage 2 is activated, the seat heating is switched off by pressing the Button "Heating Stage 1 "/Heating Stage 2". SW
CR39 6 When heating stage 1 is activated, the system switches to heating stage 2 when the Button "Heating Stage 2" is pressed; SW
CR39 7 when heating stage 2 is activated, pressing the Button "Heating Stage 1 " causes the system to switch Back to heating stage 1. SW
CR39 8 The seat control unit incorporates time control for the seat heating. SW
CR39 9 When stage 2 is switched on, it remains activated for five minutes. After that, the System automatically changes down to stage 1. SW
CR40 When stage 1 is switched on, it remains activated for 30 minutes; alter SW
Username: braunpe
Exclusive edit mode

Figure 2: Requirements for the seat heating

Beside these control-steps ensuring that every software requirement is fulfilled by some model-element, the management of tests was done similarly. So for some interesting requirements a test-case was written. These test-cases were specified graphically by EETs, a variant of MSCs used in AUTOFOCUS. So it can be seen, if there are test-cases for a specific requirement. For every test-case the requirements leading to a test-case can be shown. In this case-study the test-cases and the model are developed independently by different developers. An example for a test-case which is linked to a requirement is shown in Figure 4.

3.2 Design

The model is described using simple data types for internal and external messages of the system. System structure diagrams are used for the description of the architecture of the system. The behavior is described using auxiliary functions dealing with data, and state transition diagrams for the behavior of atomic components. Event traces describe interactions within the system. Event traces are generated during simulation.

The model covers all requirements as far as this has been possible from the short and imprecise specification.

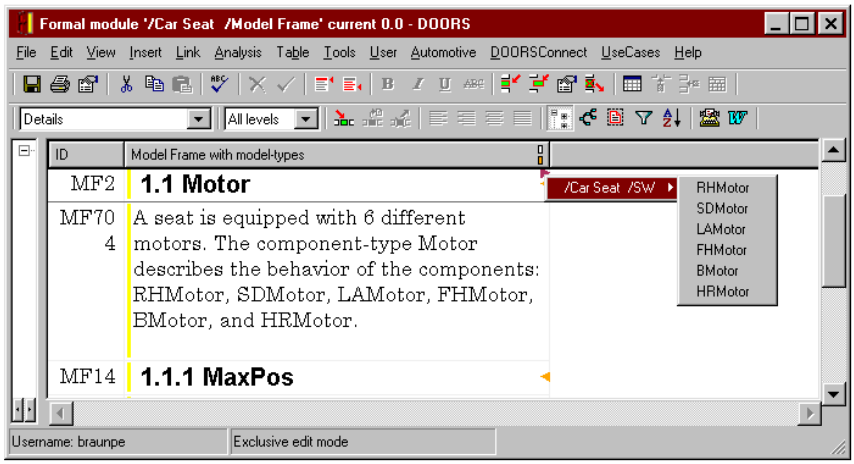


Figure 3: Model Frame

3.2.1 Interfaces

The design of the model has been started from the given interfaces. In order to apply our method to the given interfaces, the interfaces are transformed into a protocol definition that describes the interface values of the system model.

The interface of the model is described in Figure 5. We used the following DTDs to define the interfaces:

```
data SeatSwitches = LAfwd | LArev | LAstop
  | RHup | RHdown | RHstop
  | SDfwd | SDrev | SDstop
  | Bfwd | Brev | Bstop
  | FHup | FHdown | FHstop
  | HRup | HRdown | HRstop
  | Mdown | Mup | M1down | M1up | M2down | M2up
  | Heat1pressed | Heat2pressed;

data CarEnvironment = setDoorOpen | setDoorClosed
  | setClamp15Off | setClamp15C | setClamp15R | setClamp15
  | setClamp15x
  | setSpeed(Int) | setVoltage(Int);
```

3.2.2 Structure

The structure of the system is derived from the requirements that have been structured into functions. The system has two main functions: one for the motors, and one for the seat heating. The structure refines the interface model of Figure 5 and is described by the SSD

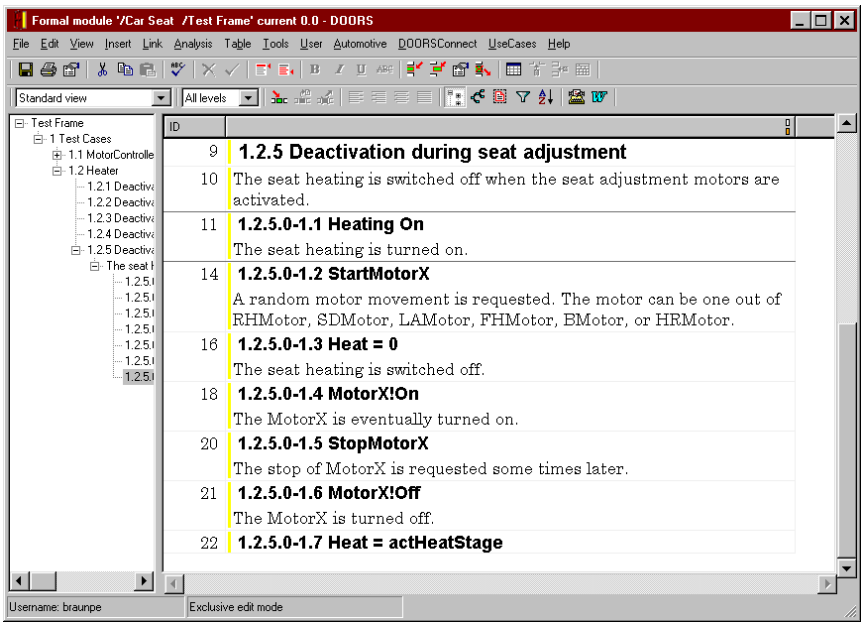

Formal module '/Car Seat /Test Frame' current 0.0 - DOORS
File Edit View Insert Link Analysis Table Tools User Automotive DOORSConnect UseCases Help
Standard view
All levels
Test Frame
1 Test Cases
1.1 MotorControlle
1.2 Heater
ID
9 1.2.5 Deactivation during seat adjustment
10 The seat heating is switched off when the seat adjustment motors are activated.
11 1.2.5.0-1.1 Heating On
The seat heating is turned on.
14 1.2.5.0-1.2 StartMotorX
A random motor movement is requested. The motor can be one out of RHMotor, SDMotor, LAMotor, FHMotor, BMotor, or HRMotor.
16 1.2.5.0-1.3 Heat = 0
The seat heating is switched off.
18 1.2.5.0-1.4 MotorX!On
The MotorX is eventually turned on.
20 1.2.5.0-1.5 StopMotorX
The stop of MotorX is requested some times later.
21 1.2.5.0-1.6 MotorX!Off
The MotorX is turned off.
22 1.2.5.0-1.7 Heat = actHeatStage
Username: braunpe
Exclusive edit mode

Figure 4: A textual specified test-case

shown in Figure 6. Since the heating shall be switched off if the motors are running, there are channels from the `MotorController` calculating if the motors are activated to the `Heater`. The system environment is used by both functions.

We start with a description of the `Heater` (see Figure 7). The `Heater` model consists of several components. The core component is `HeatControl`, it contains the controller that reacts on messages from the environment and from the panel. This component interacts with a timer for time control (`HeatTimer`). Furthermore there are additional checks (performed by other components). If one check fails a signal is passed to the merge component `MergeCheck`). `MergeCheck` combines these signals (logical or) together with the Motor-Commands that also switch of the heating. The arbiter `HeatArbiter` switches the heating off, if some checks fail. If the checks are OK, the arbiter switches the heating on to the last (or current) value. `HeatCheck15R` checks the position of Clamp15.

The more interesting part is the description of the motor controllers. Several design patterns for embedded modeling have been applied to design the structure of the motor controller:

- FlipFlop: translates button events into states and send them to other components constantly.
- Scheduler: ensures that at most two motors are running in parallel, and that at most one motor receives a command at a time.

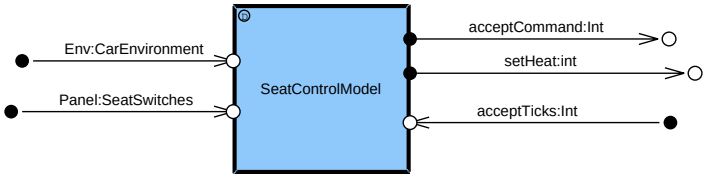


Figure 5: SSD of the System Interface

- Messaging: instead of global variables messages are used from memory components to the relevant components.
- Splitting: the acceptCommand signal is split into several signals for the different motor components.
- Merging: if several commands (or other signals) for one component are present, they are merged according to their priorities.
- Local timing: for the modeling of real-time behavior timer components are used where they are needed. Other possible variants are the use of a global time or an external time.

These modeling patterns (and others, for example for cryptography) have been developed during many projects with AUTOFOCUS.

In this paper we do not show the whole system, but for space reasons we only describe some components more detailed.

3.2.3 Behavior

The behavior is described by numerous state transition diagrams for each atomic component. For similar components STDs are reused several times. AUTOFOCUS allow to assign to each component a behavior (STD), such that STDs, can be reused. This feature has been used heavily, because six motors six controllers (and six timers, etc.) have been modeled.

The behavior of the timer is shown as an example in Figure 8.

The real-time behavior of the timer is ensured by using the system clock (instead of the variable `CurrentTimeMillis`).

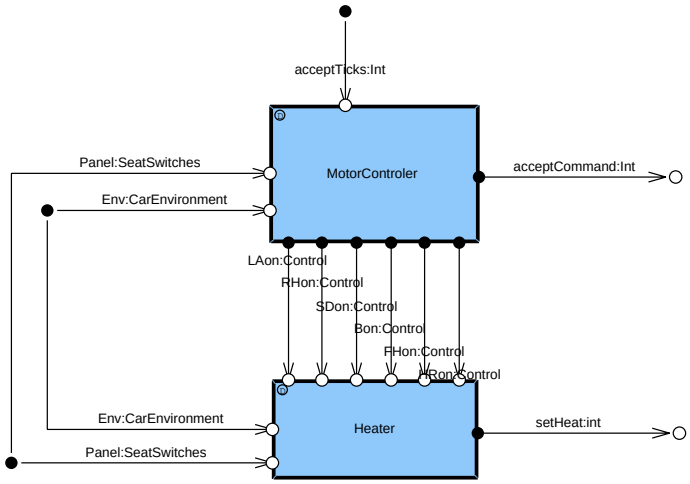


Figure 6: Structure of `SeatControlModel`

3.3 Code

The code consists of a generated model, and a manually implemented wrapper class that implements the given interfaces. For testing we added two other classes: one for the system environment (that implements `CarEnvironment`) and one for running the test. We used threads for running them independently.

3.3.1 ValidasSeat

Since neither the simulation, nor the prototyping code implements the given seat interfaces a simple wrapper has been designed, and manually implemented (ValidasSeat.java). The wrapper initializes the model and implements the methods by simply sending the events to the system. Since AUTOFOCUS has a synchronous execution model, it has to be ensured that no messages are lost when passing them to the model. Therefore we used a asynchronous model wrapper around the synchronous model. This model wrapper has synchronized input queues and passes the values to the core model (of course this wrapper is also generated).

The interfaces have been modeled with data types for all possible method calls. For example the data type `SeatSwitches` has (among others) the following values:

```
data SeatSwitches = LAfwd | LArev | LAstop
                  | SDfwd | ...   | Mup;
```

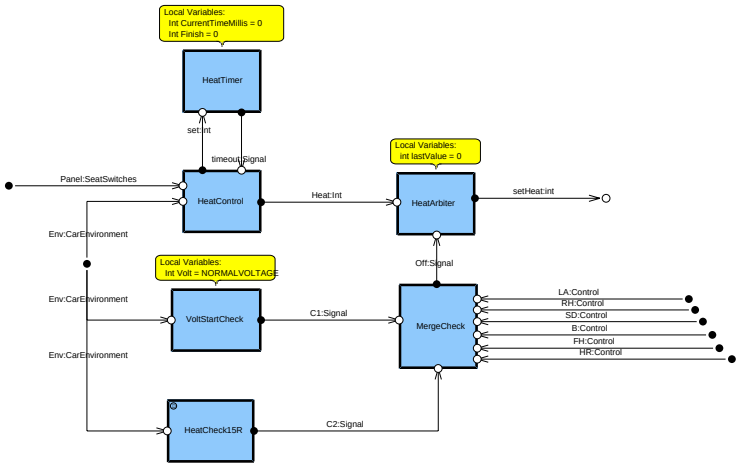


Figure 7: Structure of `Heater`

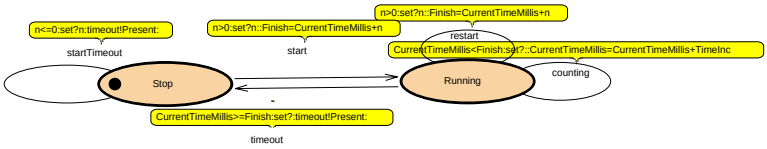


Figure 8: Behavior of Timers

These values are passed to the system every time the corresponding method is called. The return values are processed similar.

4 Validation

In this section we describe the applied validation techniques. Within the given requirements there are no quality challenges like for example:

- ensure that every user input has an effect,
- ensure that no signals are lost (due to nondeterminism),
- ensure that the response time for a motor is below 0.1 s,
- every state in the description is reachable and tested.

We do not concentrate on the validation according to those requirements, even though the Validas Validator supports the validation of critical properties using advanced methods

[Slo98]. We also don't use the generation of test cases according to different coverage criterion, in spite we use some manually specified test cases from the requirements specification.

The applied validation techniques are:

- consistency checks,
- determinism checks,
- graphical simulation of components and the whole system,
- testing the model (textually and in batch mode),
- testing the system via the specified interface and a simple GUI, and
- requirements tracing (to the model and the tests).

In this section we describe some test results and test procedures.

4.1 Improved Models

Several errors have been detected and corrected, the most interesting error was found during simulation of the complete system. It showed an integration error of the heater and the motor controller due to the switch off signals from the scheduler: The scheduler does send alternating commands to motors of group 1 and group 2. These signals are used to switch off the heating during motor movement. The result of this toggling signal was a toggling heating signal. A simple delay in the merge component of the heater fixed the error.

4.2 Consistency Checks

Several copy & paste errors have been found using the consistency checks, especially during bottom-up operations, when additional features have been added.

4.3 Simulation

AUTOFOCUS models are complete, (even if they do not allow to import code for components, actions, etc.). This allows to generate code for products, testing and simulation from the models. Simulation runs graphically.

With simulation the dynamic behavior can be evaluated. The method requires to test all requirements to components and to the system. For example the requirement that the heating shall be switched off, if motors are running (see Fig. 9) is tested by entering

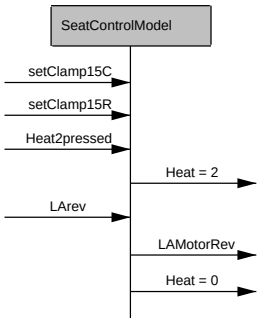


Figure 9: Heater Requirement

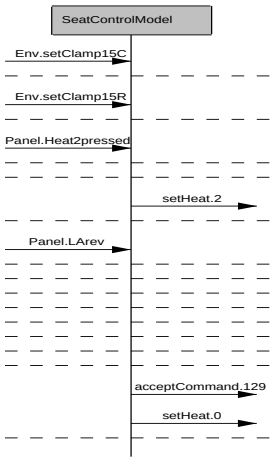


Figure 10: Protocol of Heater Simulation

the commands into the simulation environment. The result is a simulation protocol that contains the port names for the input values, the concrete values (also for the output), and the ticks (timing information represented by dashed lines). Figure 10 shows the protocol.

In addition to the system view also inner views can be animated. For example for developers it might be interesting to see if the timer for the heater has been set to the correct value. AUTOFOCUS simulation also generates EETs for the subcomponents.

The real-time behavior depends on the time required for a single step (tick) on the concrete system. For the graphical simulation the Timer can be configured via the constant `TimeInc` in the DTD `Misc`. The value of the constant represents the amount of time (in ms) the timers are incremented each step. This allows a flexible simulation of the real-time behavior.

4.4 Deterministic Check

Since we are working with a synchronous hardware oriented model, some events can occur simultaneously. In components with several inputs this can cause nondeterministic situations. Furthermore there are no message queues in the semantics, such that messages can get lost if they are not processed. The determinism check of the Validas Validator helps to detect such situations. For example in the Controller of the heat component the some nondeterministic situations have been detected (see Figure 11). Since the timeout of the timer does not occur frequently, it is very improbable that this error would have been found during the simulation (Note that the simulation of AUTOFOCUS also detects nondeterministic situations if they occur).

In a similar way it is possible to check completeness. For safety critical systems it is important to ensure that all messages are processed. This can be done using the Validas

Figure 11: Nondeterministic Situations in the Heater

Validator. For example it could be detected that certain states do not process timeout interrupts.

4.5 Testing

There are several forms of testing the model, the simplest one is to simulate the system interactively (see Section 4.3). Next step is to simulate the generated code without graphical animation (textually). The Validas code generator supports this form of testing with an interactive code that can be executed.

4.6 System Test

In order to apply an overall system test, a graphical interface has been build (see Figure 12). The interface uses the specified interfaces (seat_interface) and visualizes the state of the tests.

5 Conclusion

The component oriented approach with the synchronous model is working fine. The hierarchic description techniques are very helpful in keeping the system modular (all state transition diagrams have less than ten states).

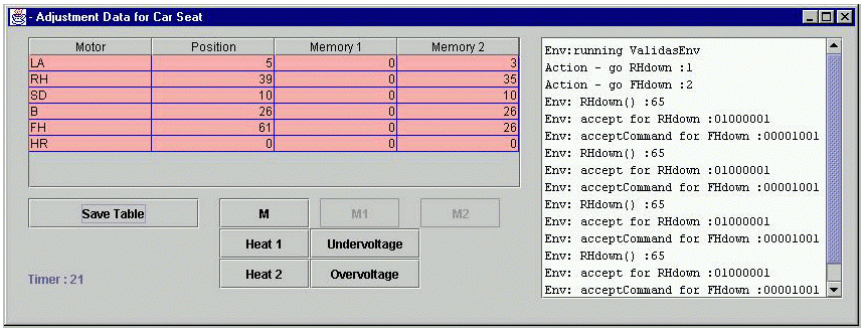


Figure 12: Test Environment

The generated code is well suited for real-time applications, however the size and run-time of the Java-Code can be improved. The generated C code is static and efficient. Reusing of parameterized functions for reused components keeps the code small.

We thank Bekim Bajraktari, Martin Rappl and Bernhard Schätz for many interesting discussions during the work on this paper.

References

[AF-02] AUTOFOCUS Homepage. http://autofocus.in.tum.de, 2002.

[Baj01] Bekim Bajraktari. Modellbasiertes Requirements Tracing. Master's thesis, Technische Universität München, 2001.

[Chr00] Daimler Chrysler. The Challange: Seat specification, 2000. Internal paper.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[HSS96] F. Huber, B. Schätz, and K. Spies. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme . In Ulrich Herzog Holger Hermanns, editor, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsbereichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9.

[Slo98] O. Slotosch. Quest: Overview over the Project. In D. Hutter, W. Stephan, P Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998.

[Tel02] Telelogic Homepage. http://www.telelogic.de, 2002.

[UML99] *OMG Unified Modeling language specification*. http://www.omg.org, 1999.

[Val02] Validas Homepage. http://www.validas.de, 2002.

Model-Based Design of ECU Software – A Component-Based Approach

Ulrich Freund, Alexander Burst

ETAS GmbH,
Borsigstraße 14,
70469 Stuttgart
Germany

Abstract: This paper shows how architecture description languages can be tailored to the design of embedded automotive control software. Furthermore, graphical modeling means are put in an object oriented programming context using classes, attributes and methods. After a survey of typical automotive requirements, an example from a vehicle's body electronics software shows the component based architecture. Introducing the concepts of component and connector refinement provide means to close the gap between system theoretical modeling and resource constraint embedded programming practice, leading to an object-oriented behavior description on the one hand and to a common middleware on the other.

1 Introduction

Around ninety percent of vehicle innovations are mainly driven by electronics. Hence, software- and systems-engineering becomes a crucial discipline which vehicle manufacturers and their suppliers have to conquer. Automotive software runs on so-called Electronic Control Units (ECU). Besides a microcontroller and memory, an ECU consists of power electronics to drive sensors and actuators. The software implementing control algorithms combines the sensor values and calculates some meaningful actuator signals. ECU software- and system-engineering is characterized by the following characteristics:

- The software is embedded which means it directly interacts with sensors and actuators and does not change its purpose during lifetime.
- The software fulfils a dedicated control task, i.e. the performance of the control algorithm imposes real-time constraints on the software to be fulfilled.
- The software is realized as a distributed system. The information of sensors located on other ECUs can (and hence will) be used to improve the control algorithm's performance. This means that the sensor information has to be sent to several other ECUs.

- The development itself is distributed. As a rule, a vehicle manufacturer employs several suppliers to deliver the control algorithms and the ECU. Since both the algorithms as well as the ECUs have to work together, the vehicle manufacturer has to do a lot of integration work to get the vehicle on the road.

Since vehicle manufactures traditionally put their focus on production cost rather than on development cost, the sensors and actuators along with the bare ECU represent almost the whole amount of costs for electronics spent. Though software does not have a direct amount of production cost, it is not for free! The only parameter where software contributes directly to production cost is by memory size. It is therefore a must for a software to be as tiny as possible. Furthermore, there is a direct relationship between the amount of production cost for sensors/actuators and the complexity of the control software in between.

To meet these constraints several ECU-programming techniques have been identified in the last twenty years:

- Establishment of building blocks and sub blocks. Exchange of information between building blocks asynchronously by means of messages (ECU global variables) instead of synchronous procedure calls.
- Call stack minimization due to limited synchronous function calls within a building block.
- Use of message duplication in case of tentative task interruption. Cyclic tasks with well chosen cycle time ensure the meeting of hard real-time constraints.
- To save overall vehicle's weight and wiring harness, bus-systems are preferred for the communication between ECUs.
- In case of bus-interconnected ECUs, cyclic broadcasting mechanisms with collision resolution (e.g. CAN) are the preferred communication means.
- Explicit and static scheduling of functions within building blocks according to the timing requirements keeps track of the memory demands and – more importantly – gives the software engineer the chance to intervene.

Well established analysis and design methods in computer science on the one hand and control engineering on the other are UML [OM99], SA/RT [WM85] as well as static data flow graphs [LP95], the latter are better known as control engineering block diagrams. These methods serve well the analysis phase but clearly lack the design requirements of ECU software:

- In UML class diagrams it is not possible to specify communication constraints in associations.
- Communication between or within an (orthogonal) StateChart [Ha87] is by means of events – the order of event handling has to be specified elsewhere to ensure hardware constraints.

- The strength of static data flow diagrams, the automatic built schedule of functions within building blocks and hence invisibility to the user, is their weakness too. To fit a building block into an ECU explicit scheduling of functions has to done elsewhere.

Even worse, all these methods allow a complex design by employing techniques not suited for a efficient ECU software design (e.g. orthogonal states).

This paper is organized as follows: Section 2 introduces abstraction levels in automotive control software engineering. A typical architecture description language for automotive purposes is described in section 3 and used throughout this paper. This language is further refined by behavioral classes (section 4) and component instances (section 5). Section 6 brings separate functions in a vehicle context.

2 Abstraction Levels in Automotive Control Software

Automotive Control Software can be viewed from different levels of abstraction. During the analysis and design process, new design and implementation information will be added to an analysis model and then transformed into ECU software [SZ02].

An appropriate modeling language will cope well with all levels of abstraction, acting as an information integration tool. Typical abstraction levels are

- the analysis model,
- the design model (functional architecture model),
- the implementation model (physical architecture model)
- and the software itself.

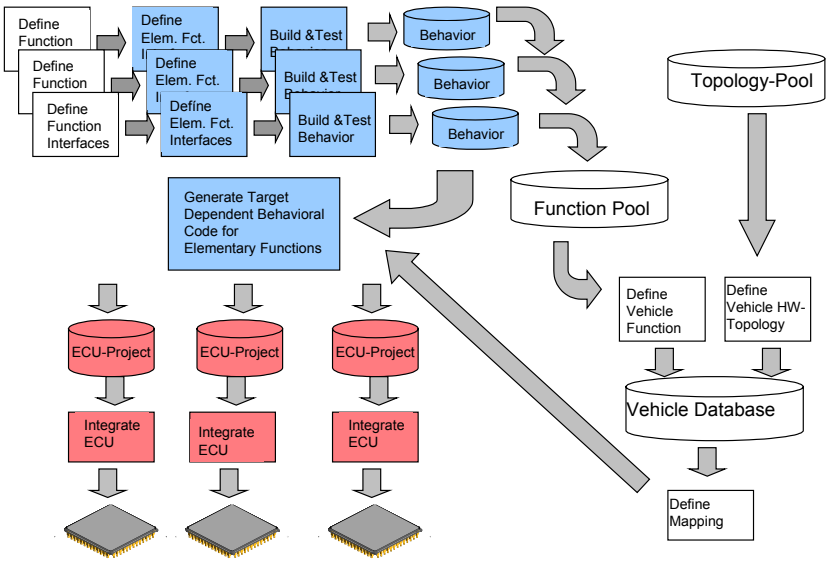


Figure 1: Two-stage design process

On every level of abstraction, it is possible to simulate the model, to check the properties of the model and to generate code for an appropriate target, e.g. a PC, an experimental hardware or a series production ECU. Generally speaking, the modeling language is capable of linking the information horizontally (e.g. simulation and code-generation) and vertically (i.e. between different layers of abstraction). This approach forms the basis for the development according to the V-Cycle. However, abstraction levels do not focus on the other side of the pie, namely the interaction between the vehicle manufacturer and its suppliers.

As mentioned above, the vehicle manufacturer is responsible for the overall functionality whereas the suppliers deliver the control algorithms and the hardware. It is the task of the vehicle manufacturer to coordinate the suppliers and let them work as cohesive as possible together. Means for identifying and describing vehicle control functions are necessary; specifying the interfaces is crucial. Currently, the interface description is more or less the communication matrix of a CAN-Bus, i.e. a list of how to link application signals with CAN frames. Due to massive integration problems, it is common understanding among the vehicle manufactures that the bus system is the wrong level of abstraction – some higher level means for integration are necessary.

Provided the appropriate means are available the development of a vehicle control function can be divided in two separate stages. The vehicle project independent development of functions and their tailoring to dedicated vehicle projects.

The vehicle manufacturer identifies vehicle functions, the interaction of elementary functions within the function or with other functions. The vehicle manufacturer asks suppliers to deliver vehicle functions which might be demonstrated by rapid development systems or simulation. The quality and performance of the functions might be assessed – function suppliers might be candidates for future series production projects.

Driven by the market requirements the vehicle manufacturer eventually decides to start a series production project (new vehicle). Instead of reinventing all functions the vehicle manufacturer asks a dedicated supplier to deliver its function for the project. The vital step of identifying the functions has been done independently. All elementary functions are mapped to the ECUs being involved in this project. Needless to say that the communication of the elementary functions between ECUs determines communication matrix. The supplier now receives the mapping system, puts its algorithm in the elementary functions and generates the code for the given ECU.

Figure 1 shows the different tentative roles of the vehicle manufacturer and the supplier. The white boxes show the specification and integration activities of the vehicle manufacturer whereas the rest of design tasks is normally done by the suppliers. The upper left parts leading to the function pool are done independently of a vehicle project. The lower parts are vehicle project dependent.

The next sections presents appropriate means for identifying functions, elementary functions and interfaces. Furthermore, the modeling of the behavior and its mapping to tiny runtime-systems will be described.

3 Component Based Modeling of the ECU-Software Architecture

3.1 Body Electronic Example

A simplified software controlling a window lifting facility shall demonstrate the concepts of architecture modeling and the subsequent refinement steps necessary to run the software on an ECU-network.

The control-software evaluates the state of a switch and drives a motor which opens or closes the window. Of course, opening and closing has to be stopped when the lower or upper limit is reached w.r.t. to the vehicles body. An anti-squeeze-function[1] is omitted due to complexity reasons.

The function offers a normal open/close mode, i.e. the button is pressed during the whole process. A 'tip-mode' opens or closes the windows by pressing the switch only for a very short time. The window opens or closes until either the limit is reached or the button is pressed again. Limit detection is based on measuring the window-lifter's motor current.

3.2 Architecture Description Languages

As a general perception, software architecture is often described by means of "box-and-line" diagrams. Though being very popular they have the big disadvantage that their correctness can neither be ensured by construction nor later be checked by formal methods. Architecture Description Languages (ADL) try to keep the advantages of "box-and-line" diagrams, i.e. their simplicity and understandability even by non-computer-scientists, and augment them with means to analyze their correctness. According to [Ga01], a typical ADL consists of

- Connectors
- Components
- Systems
- Properties
- Style

There exist a lot of activities to tailor architecture description languages to automotive needs. For example, in 1994 the TITUS-project [Ei97][Mü99] was started by DaimlerChrysler. This is an interface based approach [Fr00] and resembles in many cases to the ROOM [SR98][Ru99] methodology, but differs considerably in details mainly to make an 'actor-oriented' approach suitable for ECU-Software. A detailed comparison between the TITUS- and the ROOM methodology is given in [HRW01]. Projects focussing on similar aspects are the BROOM methodology of BMW [Fu98], the French AEE research effort [Bo00], and the Forsoft II (Automotive) project [GR00]. The

[1] Einklemmschutz in German

latter project expresses ADL concepts by means of standard UML and uses the tool ASCET-SD [ASD01] to bring designs down to ECU software. Last but not least, in spring 2001 the European research project EAST/EEA[2] was started as an ITEA project. One of the main goals of the EAST/EEA project is to develop a standardized ADL for automotive software.

3.3 Components and Service Access Points

The ADL described in this paper is based on the common characteristics of the above mentioned automotive ADL research efforts. Components are the basic building blocks of this ADL. They employ a class/parts/instance concept and can therefore compared with capsules in ROOM. Since instantiation can only be performed when the final runtime-system is known, component instances to come are described as parts. Systems and subsystems only consist of parts and connectors and cannot have own behavior which is different to ROOM.

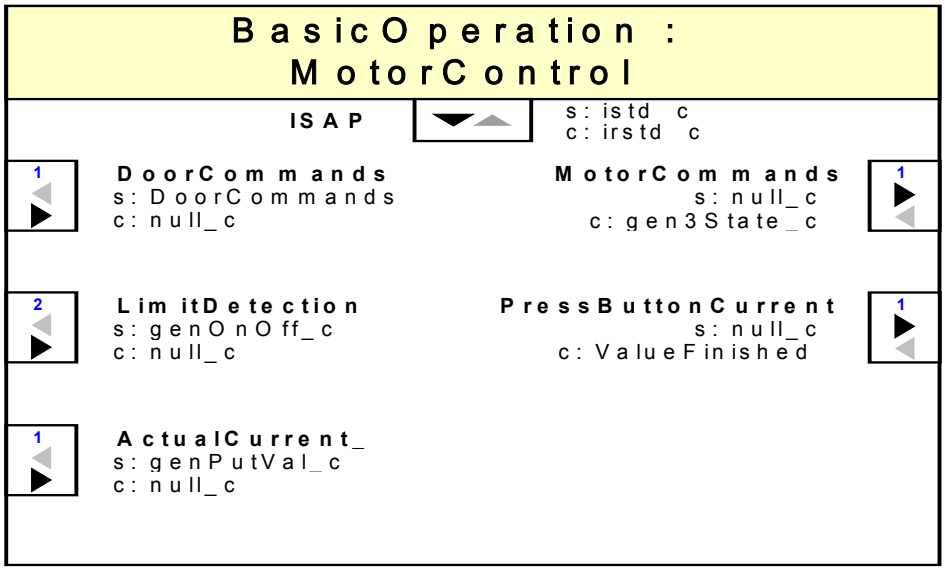


Figure 2: The BasicOperation component with its interfaces

Components are encapsulated from their environment by means of interfaces. In this ADL, interfaces are described by Service Access Points (SAPs) and ports. Interfaces describe what services a component offers to its neighbors as well as which services the component requires from its neighbors. Using a client/server interpretation of components, a SAP providing services is called server-SAP whereas a SAP requiring a service is called client-SAP. However, a SAP's role is associated to the primary communication role since SAPs can describe a bi-directional communication. Thus, every SAP employs a client and a server signal-set complementing each other. This is

[2] Embedded Architecture Software Technology/Embedded Electronic Architecture

indicated in Figure 2 by an *s:* prefix (for server) or an *c:* prefix (for client) in front of the signal-set name. If a SAP implements only one role the - non-existing - complementing signal-set is indicated by the `null_c` signal-set. The primary role of a SAP depends on which side of the component the SAP resides: left side for server-SAPs, right side for client-SAPs.

Figure 2 shows the component doing the basic motor control algorithm for one window. It provides services for handling the commands of the switch for a door, appropriate handling if the window reaches the lower limit of the door frame or the upper limit of the roof as well as means to use the actual motor current being measured by dedicated neighbor components. Server-SAPs are drawn on the left side of a component. Consequently, the required services of the component are shown on the right side which are requests to drive the motor in the appropriate direction and delivering a maximum current, measured during a given time. Using SAPs only on the left or right side is one further difference to ROOM[3] and reflects the data flow thinking of automotive control engineers.

Whereas SAPs describe the functional interface, ports are used for navigation purposes between components. Especially, the number of ports per SAP indicate how many clients (or servers) can use the provided service. The service itself is the same for all ports. From this point of view, ports can be interpreted as instances of a SAP. Every SAP must have at least one port. Subsequent sections will show how ports relate to the middleware and SAPs to the behavior of a component.

For example, the `DoorCommands` SAP of the component `BasicOperation` has one port, indicated by the number at the top of the SAP symbol. The `LimitDetection` SAP below the `DoorCommands` SAP has two ports, one for the upper- and one for the lower-limit. Both ports will convey the signals of the signal-set `genOnOff_c` depicted in

Figure 3.

A port of a client-SAP can be connected to more than one component, but it depends on the communication mode whether all connected components will receive a request or only dedicated components. Two communication modes are possible:

- peer-to-peer, meaning that the port has to be selected separately or,
- broadcast, where all connected classes will receive a request.

The broadcast mode is typically used for signals being used by several clients in the vehicle, e.g. the vehicle voltage, the clamp-state or the vehicle's actual velocity. As a rule, the peer-to-peer mode is used to drive several devices of the same type, e.g. an LED.

[3] The ISAP service access point on the top of Figure 2 is only used initialization purposes

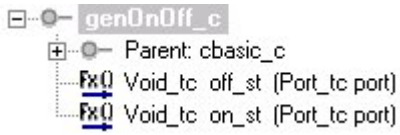


Figure 3: Signal-set consisting of the on and off method

3.4 Kinds of components

Components are elementary in the sense that they do not have further decompositions. Components can be distinguished into client/server and firmware components. Firmware components are directly connected to sensors and actuators. Their behavior can be described by means of C-code. Since firmware components directly incorporate HW-drivers, they are bound to specific ECUs. Client/server components are independent from hardware and can be assigned to arbitrary ECUs in a mapping step described later. An exception are client/server components next to firmware components performing adaptations w.r.t. to sensor and actuator peculiarities. All other client/server components are called monitors.

3.5 Connectors

In this ADL, services are described by means of methods having as a rule no return values. Therefore, the methods have the same meaning as ROOM-signals. The aggregation of all methods (or signals), offered at a server-SAP or being incorporated at a client-SAP, establishes the signal-set being transferred by a connector. Signal-sets can be structured hierarchically using a single inheritance mechanism. A “NULL” signal-set containing no methods represents the root of the signal-set tree. The functionality of a signal-set can be extended by creating a child set and adding new methods.

Figure 4 shows the signal-set `DoorCommands`. Its parent is the `null_c` signal-set set having no methods whereas the `DoorCommands` signal-set consists of the methods `off()`, `open()`, `close()`, `tipopen()`, `tipclose()` and `stop()`.

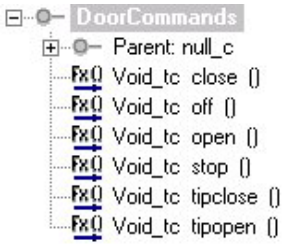


Figure 4: The DoorCommands signal-set

A connector employs two signal-sets having their roles indicated at the connected SAPs by the prefixes *s:* or *c:*. It is mandatory that a server signal-set of a component's SAP has its counterpart as client signal-set at the connected SAP of the neighbored component and vice-versa.

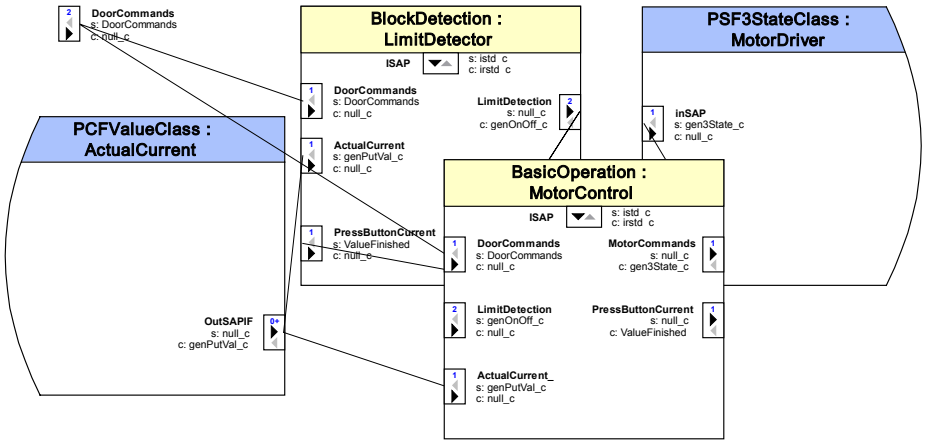


Figure 5: Inner View of the Window Lifting subsystem.

During the refinement phase, the methods have to be described by graphical or textual models (which are translated to C-code later on). This is achieved by behavioral modeling tools (BMT) or pure C-Code. In simple applications the whole functionality can be expressed within the methods, whereas in more complex designs they act as glue for the input vector of a finite-state-machine.

3.6 Systems

Systems serves the need for hierarchy. They can include components or further systems as parts and offer services at SAPs. Systems use services of other sub-systems or components. Since the SAP of a system represents the SAP of a component, the connection between the system's SAP and the component's SAP is called binding. Connected SAPs between subsystems are called bindings too.

The top level system describes the entire structure of the application. Since in automotive software resources are always allocated statically and dynamic instantiation is not used, all connectors are already resolved at compile time.

The example (sub-)system in Figure 5 shows the interface to the outside world in the upper left corner, i.e. the commands of the door-switch. The 'half-rounded' component on the left is used for sensing the motor's current whereas the rightmost component is used to drive the motor directly. The elementary components in the middle are the basic motor control and the limit detector. The door command signals are evaluated by both

elementary components. The same holds for the actual motor current. The measured maximum current is sent from the basic motor control component to the limit detecting component.

3.7 OSEK-based Remote Procedure Call

Using an event driven style, communication between components is asynchronous and explicit. To have control over the timing behavior of two components residing on remote ECUs, it is necessary for the designer to be aware of the traffic a remote procedure call will generate. For example, a `getValue`() procedure call has to be modeled with no return value. Whereas in the classical remote procedure call (RPC) world calling a getValue method of a (tentative stateless) server and expecting the result at a later (unspecified) point in time, the `getValue()` method in the OSEK[4]-based RPC world (or ORPC for short) can only set a flag at the server. The server will then notice the set flag and calls the `putValue(real result)` method of the client. The result will be sent as the actual parameter of the method, thus using the secondary roles of the SAPs' complementary signal-set.

This none-stateless interpretation of a remote procedure call under automotive constraints, hence OSEK-based Remote Procedure Call, not only makes the timing implications explicit to the designer but furthermore encourages a clear design based on pure interfaces. Components support this design approach.

4 Means of Behavioral Modeling

The behavior of a vehicle control function describes the functionality of the system. The system behavior will be measured against the performance criteria. During the analysis and design phase, the performance criteria of a control algorithm has to be checked by means of analysis, simulation and experiments.

Since vehicles are safety critical systems, it is wise to use system theoretic modeling means like finite state machines or control engineering block diagrams to design control algorithms. To bring the design down to an ECU, a more software oriented modeling is crucial, e.g. to make use of a class/instance concept while keeping the advantages of a graphical behavior description. A tool providing this dedicated automotive control software view is ASCET-SD.

[4] OSEK means "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" and is a standard for automotive embedded operating systems

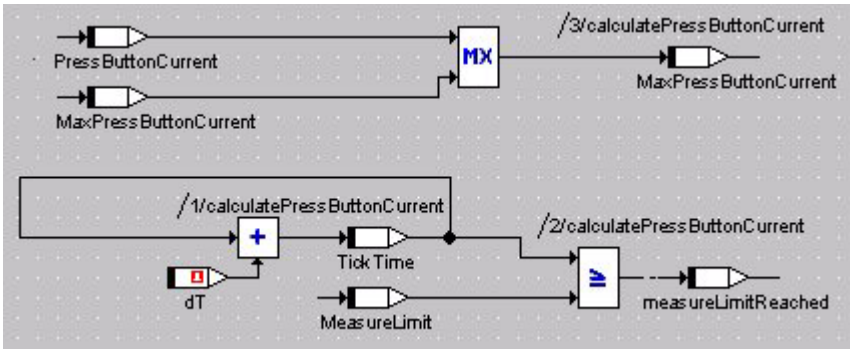


Figure 6: Data-Flow graph method for the maximum current detection

4.1 Using Behavioral Classes as Component Refinement

Component refinement means to add behavior to the components. The component's interfaces have to match the input- and output signals of the control algorithm. The class/part/instance concept of the above described ADL can only be kept during the step of component refinement by using the class/part/instance concept for behavioral modeling too. Furthermore, the method-like signals of the ADL's signal-set should have a direct counterpart in the control-algorithm. Thus, the behavioral class concept described below forms the conceptual basis for component refinement.

4.2 Behavioral Classes

A behavioral class captures control algorithms by means of methods and attributes. Inheritance and associations are omitted. Inheritance is covered by means of variants of a component. A behavioral class is a prototype and can have multiple instances somewhere in the control algorithm. It may use other classes by means of aggregation.

Within an aggregate of an object, the communication is done by means of synchronous method calls, i.e. by calling a method of the aggregated class. In a behavioral class the execution sequence of statements is given by the order of the statements. A method is a collection of statements realizing Boolean and arithmethic expressions as well as method calls to aggregated objects. Control structures like loops and selections constitute a powerful programming language.

Whereas textual description languages focus on statements, graphical descriptions emphasize the system theoretic aspects. Hence, the methods are described by either a finite state machine or a data flow graph. The methods of the SAP `DoorCommands` are shown in Figure 7. If one of these methods is called, the appropriate the enumeration type `SwitchState` will be set to the appropriate value. The method names have the form **/1/off** for the `off()` method. The number in front of the name shows the sequence number. A method calculating the maximum measured current within a given

time is depicted in Figure 6. It is named `calcPressButtonCurrent` and realizes a sequence of three graphical statements.

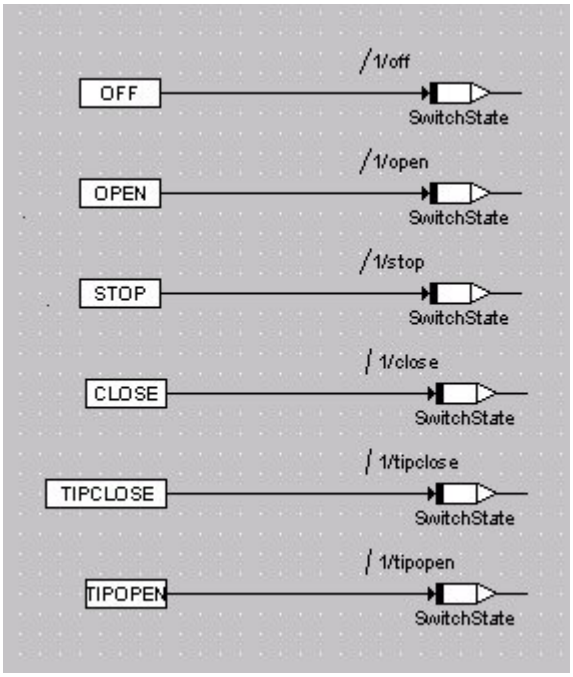


Figure 7: Data flow graph of the DoorCommands methods

Figure 8 shows how the enumeration attribute `SwitchState` is evaluated by the finite state machine diagram of the method trigger. The method `calcPressButtonCurrent` is in invoked in a refined diagram of the hierarchical state `MotorDown`.

4.3 Mapping of Signal-Sets to Methods of Behavioral Classes

The methods of a behavioral class correspond to the signals in a signal-set. Since a component maintains its signal-sets by means of service access points, the behavioral class has to implement all signal-sets in the context of a SAP. Method templates can be generated out of a component description of this ADL.

5 Means of Runtime-System Modeling

As stated in the introduction, automotive control software is tailor-made to a series production vehicle. Whereas the hardware consists of interconnected ECUs nowadays an ECU will employ tiny operating systems. An assessment of the runtime properties of an automotive control function requires its resource allocation scheme which can only be

evaluated on instance level. A component instance combines middleware aspects with instances of behavioral classes. The latter can be derived by means of component refinement whereas the former is the result of connector refinement described in the next section.

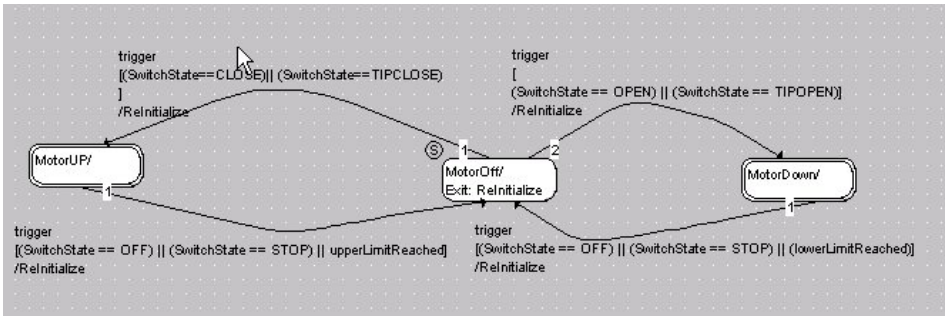


Figure 8: State Transition Diagram of an (Extended) Finite State Machine Behavioral Class

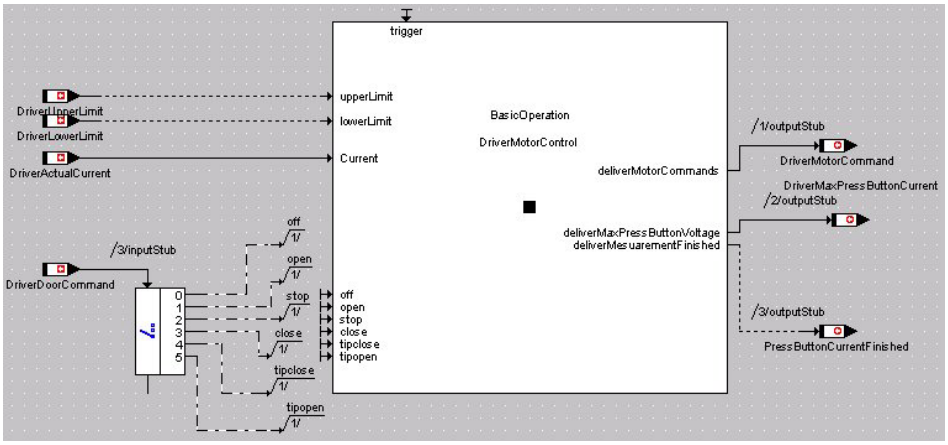


Figure 9: DriverMotorControl component instance with its surrounding middleware

5.1 Connector-Refinement

As mentioned above, the interfaces of a component are described by means of SAPs and ports. Methods of a signal-set employed at a SAP form the template for the methods of a behavioral class thus being the conceptual basis for component refinement. Ports are a template for the IPC-buffers of the middleware and therefore establish the conceptual basis for connector refinement.

5.2 IPC-Buffers

Asynchronous communication between tasks in a real-time operating system is realized by the mailbox principle[5]. Since automotive control systems have a static structure, the mailbox has dedicated entries for each communication connection realizing a connector in a typical architecture description language. Figure 9 shows an example associating the behavioral class instance `DriverMotorControl` with several mailbox-entries (IPC-buffers) on both sides of the component instance. IPC-buffers being read from are shown on the left side whereas IPC-buffers being written to are shown on the right side. The behavioral class instance in the middle has connections from its methods to the IPC-buffers. These connections are called stub-routines and may contain operators. Typical operations are endian conversions or 'method number'-interpretation described in the next section. Therefore, all graphical elements of Figure 9 not belonging to the behavioral class instance constitute the middleware contribution of the component. The ensemble of middleware contribution and behavioral class instance is called a Module in ASCET-SD. On ECU level the IPC-buffers are typically realized as global variable which might be duplicated in case of a tentative interruption by a higher priority task.

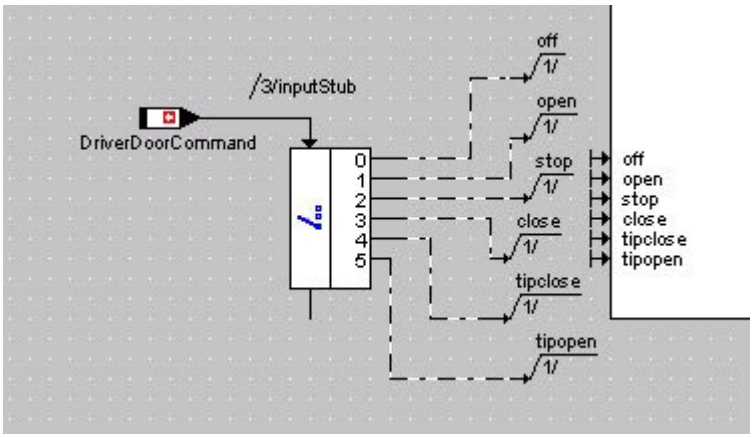


Figure 10: DoorCommands stub-routine

5.3 Stub-Routines

Reading from and writing to mailbox entries is performed by so-called stub-routines. It is their task to read values from the input mailbox entry and call the method of a behavioral class, interpreting the just read values as actual parameters for the methods of the behavioral class. In a signal-set, every method has an associated number starting with 0 for the first method. Depending on the type of the runtime-system, tentative formal

[5] The mailbox might be organized as a queue.

parameters of a method can either be stored in separate IPC-buffers or concatenated to the bits reserved for the method number.

The stub-routine for the `DoorCommands` signal-set in the lower left part of Figure 9 is depicted in more detail in Figure 10. The stub-routine `inputStub` reads the method number out of the IPC-buffer `DriverDoorCommand` and calls the appropriate method of the instance `DriverMotorControl`. Remember that the methods of the `DoorCommands` signal-set do not have formal parameters.

As written in section 3, the `LimitDetection` SAP of the component `DriverMotorControl` has two instances in form of ports. Whereas Figure 2 shows only the number of employed ports at the top of the SAP symbol the corresponding IPC-buffers are made explicit on the middleware level. The middleware contribution of the `LimitDetection` SAP is shown in detail in Figure 11.

Figure 11: IPC-buffers of the LimitDetection SAP

To summarize, a component instance consists of :

- An instance of a behavioral class
- Mailbox-entries related to the SAPs of a component
- Stub-routines

From an ECU-centric point of view, the ensemble of all mailbox-entries and stub-routines defines its middleware and hence is part of the runtime-system. The middleware of an ECU, i.e. mailbox-entries and stub-routines, can be automatically generated by using the system model built in the above described ADL. Furthermore, the instantiation process is performed automatically too.

5.4 Scheduling of Component Instances

Scheduling elements are implemented as void/void C-functions being called from OS-tasks. The order of the scheduling elements within an OS-task determines its priority within the task. The calling sequence stub-routine/method-call of a behavioral-class instance will be implemented in a void/void C-function and thus forms a scheduling element. Hence, the activation time of the OS-task determines, via its associated scheduling elements, the timing behavior of the control algorithm realized by behavioral classes.

6 The Vehicle Perspective: ECU-Networks

The above sections have described how a typical architecture description language can form the backbone of component- and connector refinement. The result of the refinement steps is a list of component instances per automotive control function. The ensemble of all component instances to be used in a vehicle determines the software architecture. In an allocation step, the component instances are mapped to ECUs, i.e. forming a deployment diagram. After this mapping, some signals have to be exchanged via a PDU[6] (e.g. a CAN-Frame) of the ECU-network. All PDUs are defined w.r.t. the vehicle. Mapping of every connector to a single PDU is not feasible in an automotive environment because of resource- and timing constraints. Hence, it is common sense to map several signals of connectors to a single PDU provided that they own the same timing properties. Addressing is not an issue because the CAN-bus uses broadcasting mechanisms. The list of signal-sets to be transmitted over the communication medium is given by the distribution of the components to the ECUs. The communication system can be validated by means rate-monotonic-analysis and simulation. Furthermore, the communication software can be configured automatically out of a component instance/ECU mapping description.

7 Summary

To enhance the productivity in embedded automotive control software design, a clear software architecture is indispensable. A typical ADL forms the backbone of a vehicle's software architecture. Components are refined by means of behavioral classes whereas connectors are realized by well established ECU-programming means. Hence, an appropriate refined ADL constitutes the conceptual basis for model-based design of distributed ECU-software.

References

[ASD01] ASCET-SD User's Guide Version 4.2; ETAS GmbH; Stuttgart; 2001.

[Bo00] Boutin, S.: Architecture Implementation Language (AIL); 1er Forum AEE; Guyancourt; March 2000; http://aee.inria.fr/forum/14032000/SB_Renault.pdf.

[Ei97] Eisenmann, J. et al.: Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client Server Architektur; VDI Berichte (1374); pp. 309 – 425; 1997; (in German).

[Fr00] Freund, U. et. al.: Interface Based Design of Distributed Embedded Automotive Software - The TITUS Approach. VDI-Berichte (1547); pp. 105 – 123; 2000.

[6] Protocol Data Unit

[Fu98] Fuchs, M. et al.: BMW-ROOM An Object Oriented Method for ASCET-SD; SAE Paper 98MF19; Detroit; 1998.

[Ga01] Garlan, D.: Software Architecture; in Wiley Encyclopedia of Software Engineering,; J. Marciniak (Ed.); John Wiley & Sons, 2001.

[GR00] Gebhard, B.; Rappl, M.: Requirements Management for Automotive Systems Development; SAE 2000-01-0716; Detroit; 2000.

[Ha87] Harel, D.: StateCharts: A Visual Formalism for Complex Systems; Science of Computer Programming 8(3); pp. 231- 247; 1987.

[HRW01] Hemprich, M.; Reiser, M.O.; Weber, M.: Die TITUS-Modellierungsnotation und ihre Zuordnung zu UML/RT; OBJEKTspektrum 2/2001; pp. 32 ff.; 2001. (in German).

[LP95] Lee, E.A.; Parkes, T.: Dataflow Process Networks; Proceedings of the IEEE; vol. 83; no. 5; pp. 773-801; 1995.

[Mü99] Müller, A.: Client/Server-Architektur für Steuerungsfunktionen im KFZ; it+ti Volume 41; Issue 5; pp. 41 ff. Oldenbourg-Verlag; 1999; (in German).

[OM99] OMG: UML Unified Modeling Language Specification. Version 1.3; 1999; http://www.omg.org.

[Ru99] B. Rumpe et al.: UML + ROOM as a Standard ADL; Proc. ICECCS'99 5th Int. IEEE Conf. on Engineering Complex Computer Systems; pp. 43 - 53; F. Titsworths (eds.); IEEE Computer Society, Los Alamitos; 1999.

[SR98] Selic, B.; Rumbaugh, J.: Using UML For Modeling Complex Real-Time Systems; 1998; http://www.rational.com/media/whitepapers/umlrt.pdf.

[SZ02] Schäuffele, J.; Zurawka, T.: Automotive Software Engineering – Current Situation, Perspectives and Challenges; Automotive Electronics I/2001; pp. 10 - 21Vieweg-Verlag; Wiesbaden; 2002; (in German).

[WM85] Ward, P.; Mellor, S.: Structured Development for Real-Time Systems. Prentice-Hall, 1985.

UML Metamodel Extensions for Specifying Functional Requirements of Mechatronic Components in Vehicles

Jörg Petersen[1], Torsten Bertram[2]

Gerhard-Mercator-University
Institute of Information Technology[1], Institute of Mechatronics and System Dynamics[2]
47048 Duisburg, Germany
[joerg.petersen@uni-duisburg.de]

Andreas Lapp, Kathrin Knorr, Pio Torre Flores, Jürgen Schirmer, Dieter Kraft

Robert Bosch GmbH, FV/SLN
70442 Stuttgart, Germany

Wolfgang Hermsen

ASSET Automotive Systems and Engineering Technology GmbH, SF/EAS1
70442 Stuttgart, Germany

Abstract: Increasing demands concerning safety, economic impact, fuel consumption and comfort result in a growing utilisation of mechatronic components and networking of up to now widely independent systems in vehicles. The development of networked electronic control units (ECU) as the most frequent mechatronic applications contains three core aspects: the development of the (control) functions itself, and their realisation in hardware and software as embedded systems. A co-ordinated, systematic and concurrent function, hardware and software development process including co-engineering and simulation environments requires a detailed specification in early development phases and a formalised description to improve the clearness of these specifications, decrease contradictions and increase information density. The Unified Modeling Language (UML) offers such a formalised description facility. A UML metamodel will be presented used for a mapping of automotive domain specific functional models onto UML models including constraints formalised by Object Constraint Language (OCL) expressions. The model also comprises the specification of functional interfaces together with a hierarchical decomposition of the system. The UML automotive domain models are basis for the system design and architecture and support aspects like re-use, exchangeability, scalability and distributed development. Particular importance is attached to the implementation of the UML model in a commercial tool together with a prototype checker of OCL expressions realised in Java.

1 Motivation and Introduction

Increasing demands concerning safety, economic impact, fuel consumption and comfort result in more and more complex vehicle functions. In order to fulfil these demands, an increasing utilisation of mechatronic systems and the creation of a car wide web,

connecting the up to now usually independently working systems in a vehicle, seem to be an appropriate solution. Electronic control units (ECU) are today the most frequent mechatronic applications; in upper class vehicles already over 80 ECUs control and monitor more than 170 control functions supported by approximately 450 electric servo motors. Individual components in the vehicle, in particular sensors, actuators, communication hardware and ECUs, are usually supplied by various manufacturers. To ensure quality, reliability and safety of such a complex networking system with components supplied form various manufactures, a detailed specification of the system is essential. In the analysis phase of system development such a detailed specification should cover an accurate requirements specification, the assignment of these requirements to functional units, the description of communication relationships between these units as well as a definition of the necessary interface data to enable these communications.

Actually, several groups of car manufacturers, suppliers and universities are working on architectural descriptions of these specifications in the automotive domain, e.g. [AW99], [Ho01], [Ko01], [La01a], [To01]. In the FORSOFT II project six abstraction levels of embedded systems in the automotive domain are identified: scenarios, functions, functional-networks, logical system architecture, technical system architecture and implementation [BR01]. In each level and development phase, different special suited tools are used, and notations of the Unified Modeling Language (UML) are proposed for the more abstract levels. The UML as an international standard passed by the Object Management Group [OMG00][OMG01] seems to be appropriate for a consistent modelling support throughout the entire development process. Despite some lack of semantical preciseness [EK99], the UML is widely used in the automobile industry, last but not least because of its support by diverse commercial tools. The UML offers extension mechanisms as well as the Object Constraint Language (OCL) suited to add domain related semantics to object-oriented models. A fundamental constraint on all extensions of the UML is to be strictly additive to the standard UML semantics.

Within this work, the UML is used to describe a structural automotive domain model with respect to functional requirements [Be98]. This so-called CARTRONIC function architecture as one element of an open and modular CARTRONIC system architecture [La01b] is based on a structuring concept for all control systems in a vehicle. UML metamodel extensions for formalising the mapping of CARTRONIC models onto UML models are presented in this paper. These extension mechanisms comprise stereotypes, constraints, tag definitions and tagged values. A proper stereotype design as well as constraints formalised by OCL expressions are introduced to secure the precision and consistency of mapped CARTRONIC models. Especially the stereotypes are a very powerful feature expressing domain specific restrictions and have to be designed very carefully [BGJ99]. Compared to the abstraction levels described in [BR01] and [Be01], the functions and functional-networks layer are the main focus of mapped CARTRONIC models, not the later software architecture layers.

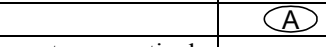
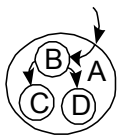
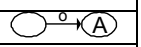
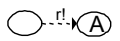
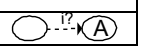
Section 2 gives a short overview of the CARTRONIC structural function architecture including a simplified torque control scenario as an example. Subsection 3.1 describes the idea of mapping CARTRONIC models onto UML models. The UML metamodel

extensions for formalising this mapping are presented in the rest of section 3. In the second subsection a hierarchy of stereotype classes is introduced, in the third the meta-modelling of communication relationships between the CARTRONIC components. Subsection 3.4 focuses on the specification of (real) time requirements for (control) functional parameters, in the last subsection the definition of OCL constraints being input for an automatic checker is introduced. The paper closes with a short summary, some hints on current work, and remarks on the relation of these UML metamodel extensions to UML-RT.

2 An Overview of the CARTRONIC Structuring Concept

The CARTRONIC structuring concept is a method to organise all control functions and systems in a vehicle. The CARTRONIC concept comprises clearly defined modelling elements as well as modelling rules to structure functional requirements in the analysis phase of system development. The result of a structuring according to CARTRONIC is a modular, hierarchical and expandable structure, which complies with these rules. The main ideas of the structuring concept are the hierarchical decomposition in functional units with defined tasks and defined functional interfaces. These functional units, so-called functional components, and communication relations between these components are the modelling elements of the CARTRONIC function structure. Functional components represent clearly defined tasks and are encapsulated by exactly defined functional interfaces. There are three different types of components: coordinators with mainly coordination tasks, operators with mainly operational tasks and information providers. The components do not automatically represent a physical unit in the sense of a constructing element or a control unit, but have to be understood as logical units. Components can be subdivided into sub-components representing the concepts of abstraction and encapsulation. Seen from outside, i.e. from the point of view of neighbouring components, a component of a function structure has to be interpreted as a system. Seen from inside, i.e. from the point of view of the sub-systems or the sub-components, the original component is only an enclosure integrating the entirety of their sub-systems.

Table 1: CARTRONIC modelling elements.

Element	***Description***	***Notation***
Component	Logical functional unit.	
System	A system consisting of several components respectively subsystems (view from inside to outside).	
Enclosure	Detailed component delegating communications to inner components expressing a part-of relationship (view from outside to inside).	
Order	Order to a component with the duty to execute a function.	
Request	Request to a component to execute a function with no obligation.	
Inquiry	Requirement for some pieces of information.	
Rule type	Structuring and modelling rules.	

The interaction of components is modelled by three different types of communication relations: order, request and inquiry. An order is characterised by the obligation to be executed by the instructed component. If the order is not executed, the receiving component has to give response to the ordering component, why the order could not be fulfilled. A request describes the demand of a source component to a target component to initiate an action or realise a function. Nevertheless, in contrast to the order there is no obligation to fulfil a request. This communication relation is used e.g. for the realisation of competing resource demands concerning power or information. The inquiry is used to get information needed for the fulfilment of an order or a request. If a component is not able to provide the requested information, it can notify the inquiring component accordingly. These communications offer a basic description of functional interfaces. Table 1 summarises the CARTRONIC elements with their graphical representations.

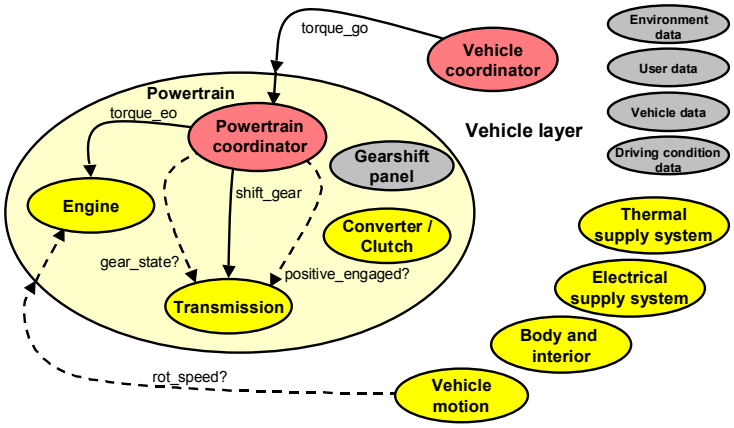


Figure 1: Example of a CARTRONIC function structure on a high level of abstraction (Vehicle layer) and the refinement of the functional component Powertrain.

Figure 1 shows an example of a CARTRONIC function structure. The Vehicle layer on the highest abstraction level contains a main component Vehicle motion, since the main task of a vehicle is the motion from one location to another. Additionally, there are three components to supply mechanical, electrical and thermal power: Powertrain, Electrical supply system, and Thermal supply system. As further components, there are Body and interior with mainly operational tasks and the Vehicle coordinator to coordinate the task and resources of the operative components. Four components serve as information providers: Environment data, User data, Vehicle data and Driving condition data. Since the complexity of these components is high, they have to be hierarchically decomposed into sub-components. As an example, this is shown for the Powertrain. It is structured into a set of more detailed components, i.e. Engine, Transmission, Converter/Clutch, Gearshift panel and Powertrain coordinator. These components have to be refined further for a more detailed description. In addition to the functional components, examples of communication relationships are given in figure 1:

- the order torque_go (to provide a certain torque at the gear output) is given by the Vehicle coordinator to the Powertrain and forwarded to the 'entry component' Powertrain coordinator,

- the order torque_eo (to provide a certain torque at the engine output) from the Powertrain coordinator to the component Engine,
- the inquiries gear_state? (present gear state) and positive_engaged? (transmission positive engaged) from the Powertrain coordinator to Transmission,
- the order shift_gear (to change the transmission gear) is given by the Powertrain coordinator to the component Transmission,
- as well as the inquiry rot_speed? (present rotational speed) from the Vehicle motion to the Powertrain, which is transferred to the responsible component Engine.

The following main features of a CARTRONIC function structure, developed in accordance with the structuring and modelling rules, can be listed [Be98]:
- defined, consistent structuring and modelling rules on all levels of abstraction,
- hierarchical decomposition of the system structure,
- hierarchical flow of orders with each component being assigned to only one orderer,
- high level of individual responsibility for each component,
- control elements, sensors and estimators are equal information providers,
- encapsulation, so that each component is only as visible as necessary and as invisible as possible for other components,
- realisation independency.

The CARTRONIC structuring concept can be used as the fundament to develop different types of vehicle and ECU configurations. It is intended to be open and neutral regarding automotive manufactures and suppliers. The resulting function structure is a basis to interconnect functions and systems of different origin, i.e. automobile manufacturers and suppliers, to a system compound. Essential for such an interconnection is the standardisation of the interfaces.

3 Extending the UML Metamodel for Mapping CARTRONIC Models onto CARTRONIC UML Models

Conceptually, a CARTRONIC model forms a structural architecture of components and connectors with respect to functionality. The mapping of a CARTRONIC model onto a formalised description is fundamental to improve the consistency of the domain model, to increase the degree of precision and specification, and to enable automated data exchanges between tools as well as transformations of models. For this formalised description the UML seems to be suited. It is a de-facto industry standard and comprises powerful extension mechanisms affecting the structure and adding semantics to user defined models by using stereotypes, constraints, and tagged values. In subsection 3.1 a consequent use of stereotypes for mapped components as well as relationships between these components is introduced. All further subsections describe the underlying extensions of the UML metamodel restricting the use of UML model elements and altogether implementing the CARTRONIC modelling rules.

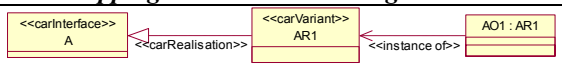
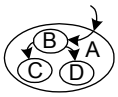
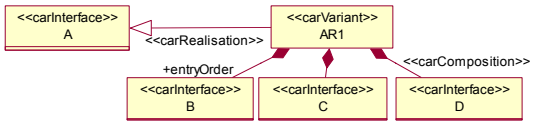
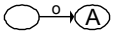
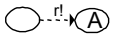
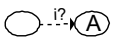
3.1 Mapping CARTRONIC Models onto UML Models and UML Objects

The mapping of CARTRONIC modelling elements onto UML modelling elements regarding the evolving UML profile for CARTRONIC is summarised in table 2. To avoid name clashes with existing UML stereotypes or well known profiles as UML-RT, most introduced classifier and relationship stereotype names are prefixed by `car`.

A CARTRONIC component A is mapped onto an (abstract) interface class named `A` (e.g. `<<carInterface>> A` in table 2). Such an interface class is used to collect all operations specifying the functions of a component, i.e. its external visible behaviour given by the CARTRONIC function structure. Each CARTRONIC interface class encapsulates the internal behaviour of this component. An interface `A` can be realised in one or more different variant classes `<<carVariant>> AR1`, `AR2`, e.g. representing and realising different physical or technical realisation principals. The UML abstraction dependency between an interface class and a realisation variant is stereotyped by `<<carRealisation>>`. Object instances like `AO1:AR1` of these variant classes ultimately represent realisations of CARTRONIC components as they are lastly implemented in ECUs. Modelling behaviour such object instances are used in UML behaviour diagrams. Finally in later software design and implementation models, most variant classes become singletons instantiated only once by a single object.

The hierarchical assignment of sub-components to a superior component is given by UML composition relationships stereotyped `<<carComposition>>` connecting the variant class, representing the superior component, with the interface classes of the sub-components. As an example, the refinement of component A onto the components B, C, and D is shown in table 2. Mapped CARTRONIC components managing all incoming orders additionally get the role name `entryOrder`.

Table 2: CARTRONIC modelling elements and their mapping onto UML.

Element	*Notation*	*Mapping onto UML modelling elements*
Component		
System Enclosure		
Order		UML-Operation with stereotype `<<Order>> o` of `<<carInterface>> A`.
Request		UML-Operation with stereotype `<<Request>> r` of `<<carInterface>> A`.
Inquiry		UML-Operation with stereotype `<<Inquiry>> i` of `<<carInterface>> A`.
Rule type		Extensions of the UML metamodel with stereotypes, relationships between stereotype classes, multiplicities, tags, and OCL expressions as constraints.

3.2 Extensions of the UML Metamodel for the Hierarchical Component Structure

In the following, CARTRONIC modelling rules are expressed by extending the UML metamodel. New stereotype classes are introduced, which are arranged in a generalisation hierarchy [OMG01, p. 2-87]. The introduced stereotype class hierarchy together with additional relationships between these classes restrict the UML mapping of CARTRONIC modelling elements. Locally as well as globally defined restrictions of CARTRONIC specific modelling rules are comprised as constraints formalised by OCL expressions.

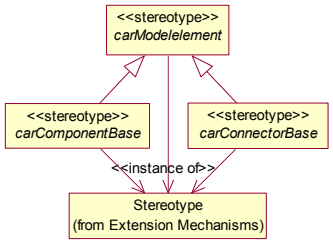


Figure 2: The root of the CARTRONIC UML metamodel stereotype hierarchy.

All newly-defined stereotype classes are instances of the UML metamodel class `Stereotype` from the `Extension Mechanism` package. Abstract stereotype classes are typed in italic names, they mainly serve in structuring the metamodel. The abstract root stereotype metaclass is called *`carModelelement`* (figure 2). Since a CARTRONIC UML model forms a structural architecture described by functional components and connectors, two principally different types of stereotype metaclasses are derived from the metamodel root, the abstract metaclasses *`carComponentBase`* and *`carConnectorBase`*.

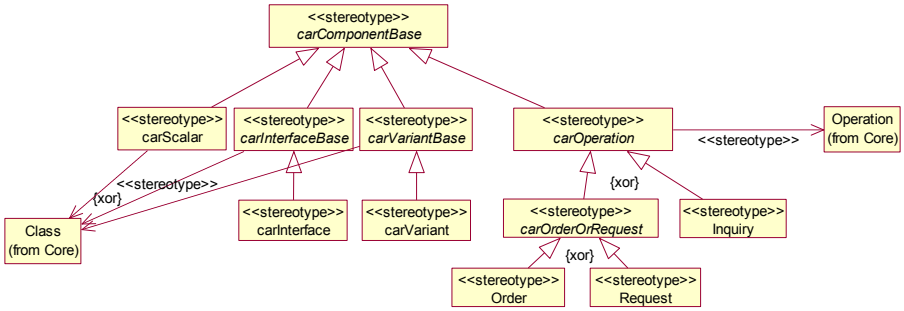


Figure 3: The stereotype hierarchy extending the UML metamodel for CARTRONIC components.

Figure 3 shows the stereotype hierarchy extending the UML metamodel for CARTRONIC components. The abstract metaclasses *`carInterfaceBase`* and *`carVariantBase`* are derived form the metaclass *`carComponentBase`* and associated to the UML metaclass `Class` from the `Core` package. From these classes the stereotype metaclasses `carInterface` and `carVariant` are derived. In later versions, additional

types of interface and realisation classes may be added. The stereotype `carScalar` for scalar values is also associated to the UML metaclass `Class`. The `xor`-constraint expresses, that only one of the three stereotypes can be assigned to a class in a CARTRONIC UML model.

The receiving component of a CARTRONIC communication relation offers a kind of functional service to the sending component. These functionalities are mapped onto UML operations in the UML `<<carInterface>>` classes with stereotypes `<<Order>>`, `<<Request>>` or `<<Inquiry>>`. In the extensions of the UML metamodel these three different metaclasses are inherited from the abstract stereotype *`carOperation`* (figure 3) respectively *`carOrderOrRequest`*. *`CarOperation`* is associated to the UML metaclass `Operation` from the `Core` package, the `xor`-constraint expresses, that only the usage of one of these stereotypes will be correct.

Figure 4: Extensions of the UML metamodel: each interface class contains at least one operation.

In figure 4 the aggregation between *`carInterfaceBase`* and *`carOperation`* specifies, that each `carInterfaceBase` class requires to have at least one offered *`carOperation`*, expressed very efficiently by the multiplicity expression `1..n`.

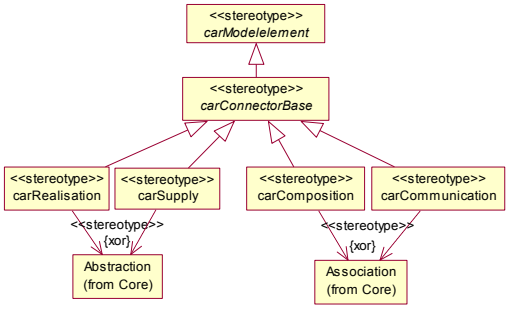


Figure 5: Stereotype hierarchy extending the UML metamodel for CARTRONIC relationships.

Connectors describe the allowed structural relationships between CARTRONIC components. For building component structure hierarchies, the two metaclasses `carRealisation` and `carComposition` are derived from the root metaclass *`carConnectorBase`* (figure 5). The stereotype `<<carRealisation>>` emphasises a UML realisation, an `Abstraction` derived from a `Dependency` relationship in the UML metamodel `Core` package. UML composition relationships are used for modelling the hierarchical assignment of sub-components to a superior component. A UML association with the stereotype class `carComposition` always connects a realising class with an interface class of a sub-component.

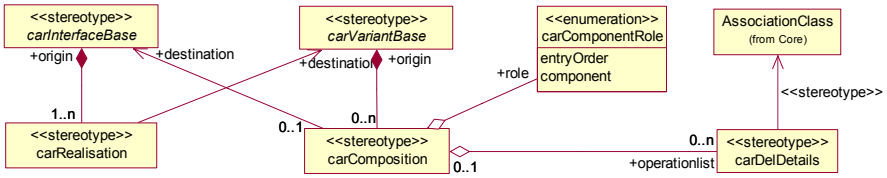


Figure 6: Hierarchy building relationships in the CARTRONIC UML metamodel extensions.

Figure 6 summarises the allowed hierarchy building relationships. They can be efficiently expressed by additional relationships between stereotype classes in the CARTRONIC UML metamodel extensions. Each metaclass *carInterfaceBase* has one or more carRealisations (multiplicity expression 1..n), but each carRealisation belongs to exactly one (encapsulating) *carInterfaceBase* (not written multiplicities are 1 by default) with public role name origin. A carRealisation connects (a *carInterfaceBase*) to exactly one carVariantBase with public role name destination at the association end of *carVariantBase*.

Each *carVariantBase* may have no (zero) carComposition relationships (a leaf in the CARTRONIC hierarchy) or an arbitrary number (multiplicity expression 0..n). Vice versa each carComposition starts from exactly one *carVariantBase* with role name origin (system/refined component). At the other end of a carComposition relationship exactly one *carInterfaceBase* is connected with role name destination (sub-component/subsystem), and vice versa each *carInterfaceBase* is part of exactly one carComposition. The multiplicity 0..1 is used because exactly one exception exists at the root of the composition tree; this exception has to be specified by an additional OCL constraint.

In the refinement of a component all operations of this component have to be delegated to the sub-components, and there has to be exactly one component as the target component for all forwarded orders called 'entry component' in the functional structure. The UML composition relationships, which are used for modelling the hierarchical structure, implicitly include the mechanism of delegation of operations respectively messages. To exactly specify a unique delegation mechanism, the stereotype class carDelDetails associated to AssociationClass in the Core package of the UML metamodel is defined (figure 6). Additionally, the unique entry component is specified based on a UML composition relation with the role name entryOrder defined in an <<enumeration>> carComponentRole. An additional OCL expression as a constraint can be specified to guarantee, that exactly one sub-component gets the role name entryOrder (see subsection 3.5).

As an example, figure 7 shows the refined functional component Powertrain from figure 1 without incoming communication relationships from other components. It is mapped onto the UML class <<carInterface>> Powertrain realised by the <<carVariant>> PowertrainR1, which is refined into the five sub-components: Powertrain_coordinator, Engine, Transmission, Converter_Clutch and Gearshift_panel, and all of them have the stereotype <<carInterface>>. They are

connected by (part-of) `carCompositions` to the class `<<carVariant>> PowertrainR1`. The `carInterface` of the component `Powertrain` offers the two operations `<<Order>> torque_go` and `<<Inquiry>> rot_speed` (compare to figure 1). The refined component `PowertrainR1` delegates `<<Order>> torque_go` to `<<carInterface>> Powertrain_coordinator` as the entry component for all incoming orders and `<<Inquiry>> rot_speed` to `<<carInterface>> Engine`. Their unique delegation is specified by the two associated classes with stereotype `carDelDetails`.

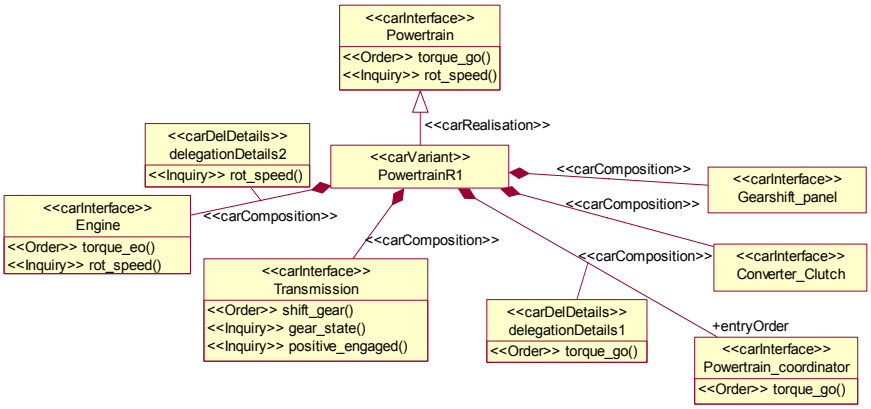


Figure 7: Example of mapping the refined CARTRONIC component Powertrain from figure 1 with definition of a unique delegation.

3.3 Extensions of the UML Metamodel for the Communication Relationships

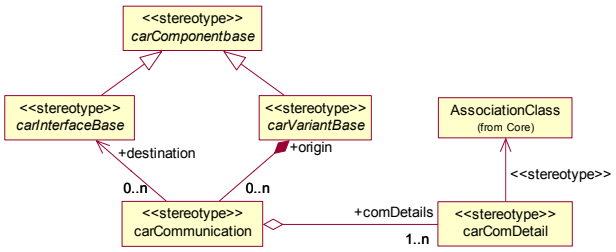


Figure 8: Communication relationships in the CARTRONIC UML metamodel extensions.

In figure 8 the allowed communication relationships are specified. The association between the stereotype metaclassses *`carVariantBase`* and `carCommunication` with multiplicity `0..n` expresses, that as much as required communication relationships from a realising component are allowed (including none). Vice versa the *`carVariantBase`* is a unique sender with role name `origin` for the `carCommunication`. A *`carInterfaceBase`* is the unique receiver in the role `destination` of a

`carCommunication` and each *`carInterfaceBase`* may be recipient of arbitrary many communication requests (multiplicity `0..n`).

Analogously to the specification of delegating an operation respectively message in a composition relationship, a stereotype metaclass `carComDetails` is aggregated to each `carCommunication`. Within these associated classes in CARTRONIC UML models, the required and used functionalities (modelled by operations in the called interface) are specified exactly, e.g. in figure 1 Vehicle coordinator orders the Powertrain to supply a certain torque_go. The metaclass `carComDetails` is associated to the UML metaclass `AssociationClass` in the `Core` Package of the UML metamodel.

Figure 9 shows the mapped CARTRONIC UML model of the example given in figure 1 including all `carCommunications` and `carSupplys` (see below). Each UML class, UML relationship, and UML operation has a stereotype defined in the CARTRONIC UML metamodel extensions. The given stereotypes for each used UML relationship clearly describe the domain specific intention of this relationship.

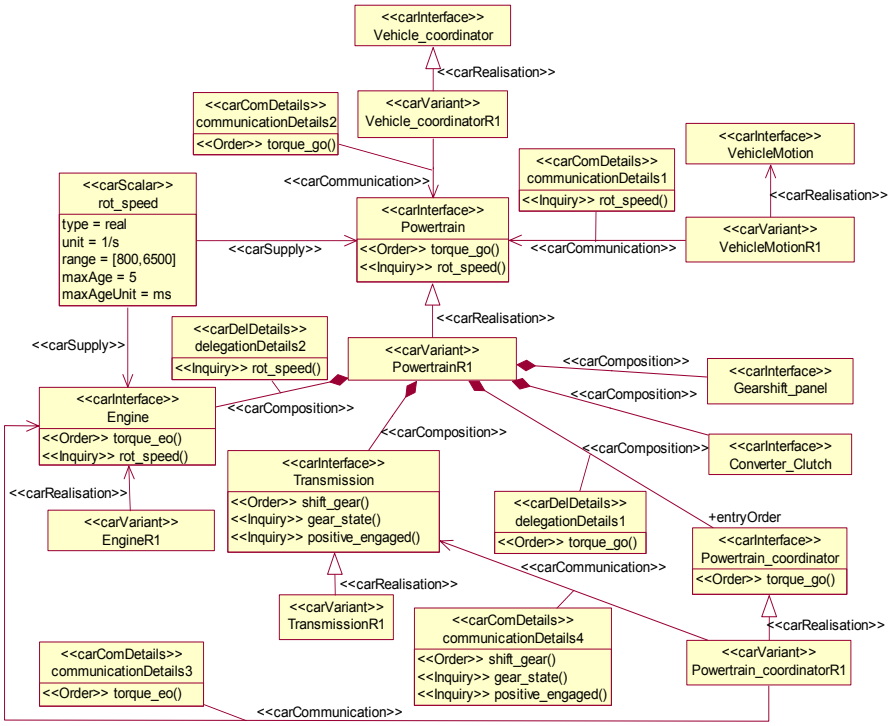


Figure 9: The CARTRONIC UML model of the given example in figure 1.

This consequent use of domain related stereotypes is very useful for an automated proof of domain model restrictions. For example, a hierarchical flow of orders according to the CARTRONIC modelling rules can be proven by a checker. In figure 9, an existing `<<Order>> shift_gear` from the `<<carVariant>> Powertrain_coordina-`

`torR1` to the `<<carInterface>> Transmission` prohibits a second `<<Order>> shift_gear` from another realising variant.

By and by combining, integrating, and enriching these UML models in the analysis phase of the entire development process will lead to more and more complete domain models being bases of the software architecture and design.

3.4 Scalar Data Interface Specifications in the UML Metamodel Extensions

For a complete function architecture, the so far discussed structural description has to be added by a behavioural specification. For automotive applications, this includes event-driven and time-driven functionality as well as (hybrid) combinations of both. Examples are wiping functionality, light switching, braking system (ABS), motor control, gear control or adaptive cruise control (ACC). Regarding especially the time-driven mechatronic systems, a data flow oriented behavioural description based on differential equations is used usually. These behavioural descriptions are essential for a consistent system development and mostly realised as simulation models in tools like Matlab/Simulink [Ma02]. By using simulation in an early stage of development, possible inconsistencies, errors and risks can be discovered and reduced or even eliminated. To use the function structure as a bases for a behavioural description, interface data have to be specified in more detail as discussed so far. This is especially relevant for the data values exchanged via the functional interfaces, which is discussed in the following.

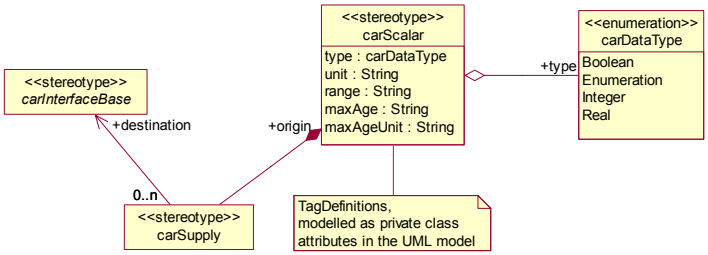


Figure 10: Tag definitions as physical data properties in the extended UML metamodel.

An inquiry for information like `<<carInquiry>> rot_speed` in figure 9 delivers an information, which is a scalar. Looking at such a scalar, it is important to specify the unit and the quality of the value. In terms of get-value-operations `carInquiry`s ask for `carScalar` values (or compounds of them) with globally agreed `type`, `unit`, `range`, `maxAge` and `maxAgeUnit` properties (figure 10). The two properties `maxAge` and `maxAgeUnit` as tags describe the supplied actualisation rate, i.e. which (real-) time requirements the providing realisation classes have to guarantee for this scalar value later on. Additionally, in the UML metamodel extensions the stereotype metaclass *`carInterfaceBase`* is related to `carScalar` via association relationships to the metaclass `carSupply` (see also in figure 5 the association between `carSupply` and `AssociationClass` from the `Core` package).

Since the interpretation of tagged values is intentionally beyond the scope of UML, the semantics must be determined by user or tool conventions; as an easy to use convention, in figure 9 the use of private class attributes instead is shown. The `<<carScalar>> rot_speed` is supplied by `<<carInterface>> Powertrain` and `<<carInterface>> Engine` via the `<<carInquiry>> rot_speed` with a guaranteed actualisation rate of `5 ms`.

3.5 OCL Expressions as Constraints for Further Domain Specific Restrictions

The extension mechanisms of the UML metamodel include the definition of constraints as domain specific restrictions. OCL expressions are used for a formalised specification of CARTRONIC specific modelling rules. These formalised rules allow an automated consistency check of fully stereotyped CARTRONIC UML models. Without such a formal metamodel only a manual check is possible.

In the following, some examples are given. OCL expression (1) specifies the CARTRONIC rule, that each component has to receive at least one order. This can be formalised by an OCL expression as an invariant for the stereotype metaclass `carInterface`. The constraint expresses, that in the set of all UML operations of a mapped functional component at least one has to have a stereotype `Order`:

```
context carInterface inv:                                    (1)
  self.carOperation->exists(self.oclIsKindOf(Order))
```

Another domain restriction defines, that all orders reaching a refined component have to be forwarded from the enclosure to exactly one sub-component with the role name `entryOrder` (see figure 6), which has to co-ordinate all incoming orders. Based on the metamodel extensions, this can be expressed as an invariant (2) for the metaclass `carVariant` counting occurrences of `carComposition` relationships with role name `entryOrder`:

```
context carVariant inv:                                      (2)
  self.carComposition->size > 0 implies
    self.carComposition->collect(role='entryOrder')->size = 1
```

Operations or attributes, which have the purpose to simplify OCL expressions describing constraints, can be defined as OCL pseudo-operations or pseudo-attributes [OMG01, p. 6-55][Wa99, p. 68, 71] serving e.g. as shortcuts in complex navigation expressions. For example, collecting all realising *carVariantBase*s for a *carInterfaceBase* is defined by (3):

```
context carInterfaceBase def:                                (3)
  let directRealisations : Set(carVariantBase) =
      self.carRealisation.collect(destination)->asSet
```

Such shortcuts are very useful defining more complex restrictions, e.g. a correct use of the connectors `carRealisation` and `carComposition` leads to a directed graph structuring the defined components. A hierarchical tree structure of the components is given, if all nodes have input degree 1 (except for the root with input degree 0) and this directed graph is cycle free.

Looking at the allowed `carCommunication` relationships, a CARTRONIC UML model defines a directed communication graph. OCL expressions can be defined to restrict the graph to communication relationships allowed by the CARTRONIC modelling rules.

4 Summary, Conclusion and Future Work

The demand for innovative and advanced vehicle functions at reasonable costs leads to more and more complex vehicle functionalities as well as an increasing and more complex networking of mechatronic components. Therefore, a detailed and systematic specification of the entire vehicle functionality will be required based on a sound semantic foundation. Semiformal models as the CARTRONIC functional structure assist in structuring and managing complexity. A formalised and tool supported model of specified functional requirements can be achieved by mapping CARTRONIC functional components and communication relationships onto UML models. Therefore, in an early analysis phase of the entire mechatronic system development process the CARTRONIC UML model is a domain model with respect to functional requirements. The mapping is based on UML metamodel extensions including stereotypes, tag definitions, tagged values, and constraints. A hierarchy of stereotype metaclasses is introduced. Relationships between these stereotype metaclasses with multiplicities and constraints given as OCL expressions represent automotive domain specific semantics regarding control systems in vehicles. Such formalised constraints allow an automated consistency check of given CARTRONIC UML models concerning the domain specific modelling rules. Choosing standard UML, different currently available commercial UML tools can be used. Up to now, neither the commercial UML tools provide a full support of the UML metamodel extension mechanisms nor support the definition of OCL expressions together with an integrated checking mechanism. Therefore, a prototype of a checker is implemented independently from the UML tool in the programming language Java. The checker is reused from the object oriented modelling concept for software of electronic control units in vehicles (OMOS) [He00]. The CARTRONIC UML models are exported in a CARTRONIC XML file format given as input for this checker. This XML export file, the CARTRONIC UML metamodel extensions, exported as an XML file too, and the specified OCL expressions are input files to the checker. At the moment, the generated output is a textual report of violated multiplicities and contraints of checked models.

The CARTRONIC functional structure assists in structuring and managing complexity in the analysis phase of the overall development process. Functional architectures influence the subsequent architectures including the software architecture. For the simulation of the dynamic behaviour of the system in early development phases, the structural description of CARTRONIC UML models can be coupled to data flow oriented ones

like Matlab/Simulink models usually used in function development. Identifying such coupling points in an incremental mechatronic system development process model based on the V-model, one focus of current work is an automated structure and information exchange based on an XML files [Kn02]. From single simulation models of domain model parts further requirement data in the entire domain model can be gained. Putting them together into one complete model comprising different variants, the calculation of global requirements, integrity as well as aspects of correctness, consistency, and completeness of a chosen variant can be proven. Such functional variants, modelled by different realising classes in a CARTRONIC UML model, require an adequate variant handling supporting re-use, exchangeability, and scalability. Continuing in the software development process, the functional architecture model is a basis for the creation of different software (design) architecture models, which obey different non-functional requirements like hardware resources, costs, etc. Software architectures influence the subsequent design, and in general, breaking down such architectures into designs is not possible in a simple straight forward manner [He99]. UML-RT is conceptualised for the design of software architectures for embedded real-time software systems [SR98] and is also used for the design of embedded systems. Three principal constructs as UML metamodel extensions are defined. Capsules are complex, physical, possibly distributed architectural objects, interacting with their surroundings through ports; capsule functionality is realised by (networks of) state machines. Ports are interface objects with an associated protocol as abstract specification of the desired behaviour. Connectors are abstract views of physical communication channels between ports. Transforming and mapping the functional architectures described in this paper onto software architectures, regarding state machine based semantics the given counterparts are capsules and classes stereotyped by `carVariant`, ports and classes stereotyped by `carInterface` or `carComDetail`, connectors and classes stereotyped by `carCommunication` or `carDelDetail`. Using a software component model to describe software designs [La01b][MA00], strategies for grouping functional components to software components have to be developed based on defined criteria like characteristics and number of communication relationships, traffic analysis, actualisation rates of supplied scalar data, and so on, leading to domain specific types of design patterns.

Acknowledgements

We thank the anonymous referees for their helpful comments.

Bibliography

[AW99] Advanced Information Technology - Workshop for Object Oriented Design and Development of Embedded Systems (AIT-WOODDES). http://wooddes.intranet.gr/, 1999.

[Be01] Beeck, M. von der; Braun, P.; Rappl, M.; Schröder, Chr.: Modellbasierte Softwareentwicklung für automobilspezifische Steuergeräte. In [VDI01], 2001.

[Be98] Bertram, T.; Bitzer, R.; Mayer, R.; Volkart, A.: CARTRONIC - An open architecture for networking the control systems of an automobile. SAE International Congress and Exposition. Detroit/Michigan U.S.A., 1998.

[BGJ99] Berner, St.; Glinz, M.; Joos, St.: A Classification of Stereotypes for Object-Oriented Modeling Languages. In [FR99], 1999.

[BR01] Braun, P.; Rappl, M.: Abstraction levels of embedded systems. OMER 2 Workshop, Herrsching, 2001.

[EK99] Evans, K.; Kent, St.: Core Meta-Modelling Semantics of UML: The pUML Approach. In [FR99], 1999.

[FR99] France, R.; Rumpe, B.: UML '99 - The Unified Modeling Language. Lecture Notes in Computer Science 1723. Springer-Verlag, Berlin, 1999.

[He00] Hermsen, W.; Neumann, K. J.: Application of the Object-Oriented Modelling Concept OMOS for Signal Conditioning of Vehicle Control Units. SAE International Congress and Exposition, Detroit/Michigan U.S.A., 2001.

[He99] Herzberg, D.: UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain. In [FR99], 1999.

[Ho01] Hofmann, P.; Fasolt, J.; Geretschläger, P.; Sakretz, R.; Wohlgemuth, F.: Automotive UML - eine neue objektorientierte Entwicklungstechnik. Published in 'Elektronik Sonderheft: Automotive', 2001.

[Kn02] Knorr, Kathrin; Lapp, A.; Torre Flores, P.; Schirmer, J.; Kraft, D.; Petersen, J.; Bourhaleb, M.; Bertram, T.: A Process Model for Distributed Development of Networked Mechatronic Components in Motor Vehicles. Accepted: IEEE Joint International Requirements Engineering Conference ‘02, Essen, 2002.

[Ko01] Kokes, M.; Querfurth, A. v.: Methodik zur Entwicklung Elektronik im Fahrzeug. In [VDI01], 2001.

[La01a] Lange, K. ; Bortolazzi, J.; Marx, D. ; Wagner, G.; Gresser, K.: Hersteller-Initiative Software. In [VDI01], 2001.

[La01b] Lapp, A.; Torre Flores, P.; Schirmer, J.; Kraft, D.; Hermsen, W.; Bertram, T.; Petersen, J.: Softwareentwicklung für Steuergeräte im Systemverbund – Von der CARTRONIC-Domänenstruktur zum Steuergerätecode. In [VDI01], 2001.

[Ma00] Mann, S.; Borusan, A.; Ehrig, H.; Große-Rhode, M.; Mackenthun, R.; Sünbül, A.; Weber, H.: *Towards a Component Concept for Continuous Software Engineering*. Berlin, ISST, ISST-Berichte 55, 2000.

[Ma02] Mathworks. Simulink, Dynamic System Simulation for Matlab. http://www.mathworks.com/products/simulink/, 2002.

[OMG00] Object Management Group (OMG): OMG Unified Modeling Language Specification V1.3, http://www.omg.org/cgi-bin/doc?formal/00-03-01, 2000.

[OMG01] Object Management Group (OMG): OMG Unified Modeling Language Specification V1.4, http://www.omg.org/cgi-bin/doc?formal/01-09-67, 2001.

[SR98] Selic, B.; Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Whitepaper, http://www.rational.com/products/whitepapers/UML-rt.pdf, 1998.

[To01] Torre Flores, P.; Lapp, A; Hermsen, W.; Schirmer, J.; Walther, M.; Bertram, T.; Petersen, J.: Integration of the ordering concept for vehicle control systems CARTRONIC into the software development process using UML modeling methods. SAE International Congress and Exposition, Detroit/Michigan U.S.A., 2001.

[Wa99] Warmer, J. B.; Kleppe, A. G.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Object Technology Series, Reading/Massachusetts, 1999.

[VDI01] VDI Tagung Elektronik im KFZ, Baden-Baden. VDI-Verlag, Bericht 1646, Düsseldorf, 2001.

A Model-Based Approach for Automotive Software Development

Peter Braun, Martin Rappl
Institut für Informatik, TU München
Boltzmannstr. 3
85748 Garching b. München, Germany

Abstract: Integrated model-based specification techniques facilitate the definition of seamless development processes for electronic control units (ECUs) including support for domain specific issues such as management of signals, the integration of isolated logical functions or the deployment of functions to distributed networks of ECUs. A fundamental prerequisite of such approaches is the existence of an adequate modeling notation tailored to the specific needs of the application domain together with a precise definition of its syntax and its semantics. However, although these constituents are necessary, they are not sufficient for guaranteeing an efficient development process of ECU networks. In addition, methodical support which guides the application of the modeling notation must be an integral part of a model-based approach.

Therefore we propose the introduction of a so-called 'system model' which comprises all of these constituents. A major part of this system model constitutes the Automotive Modeling Language (AML), an architecture centric modeling language. The system model further comprises specifically tailored modeling notations derived from the Unified Modeling Language (UML) or the engineering tool ASCET-SD or general applicable structuring mechanisms like abstraction levels which support the definition of an AML relevant well-structured development process.

1 Introduction

Within the automotive industry model-based specification techniques are becoming more and more popular allowing the complete, the consistent, and the unambiguous specification of software and hardware parts of automotive specific networks of control units. In this context model-based approaches provide methodical support to manage the integration of logical functions and the deployment of functions to distributed networks of ECUs. In addition well founded models are the source for all kind of analysis, validation, and verification activities.

A prerequisite for the design of a model-based specification methodology is a precise knowledge of the architecture of the targeted system class. In our opinion a good architecture centric language reflects issues of automotive embedded systems as modeling concepts in terms of an automotive specific ontology. Thereupon rests the construction of a system model by precisely defining relations between model elements, their classification within abstraction levels and their embedding in a development process. During

modeling all information is stored in an integrated and consistent model. To cope with the complexity of this model a system of domain specific abstraction levels provides an appropriate structuring mechanism for the specification of networks of control units on different technical levels.

The presented work is related to the work the OMG [OMG99], the U2 Partner group [Gro01], and the pUML group [CEK+00] are carrying out. In contrast to their approach we believe, that a more rigorous mathematical theory is necessary for a realistic model-based development process. Nevertheless the representation of model elements is done by specifically adapted notations from the UML 1.3 [OMG99], the ASCET-SD modeling language [WWWb], and textual specification techniques which stem from the tool DOORS [WWWc].

The paper introduces the most important features of the Automotive Modeling Language (AML). Afterwards notions of the automotive specific ontology are sketched constituting major AML modeling concepts. In the final section we draw our conclusions and provide an outlook to future work.

The work presented in this paper represent results of the research project FORSOFT Automotive - Requirements Engineering for embedded systems [WWWa]. The partners of this project are the Technische Universität München, the tool providers Telelogic and ETAS, the car manufacturers BMW and Adam Opel, as well as the suppliers Robert Bosch and ZF Friedrichshafen.

2 AML Features in a Nutshell

To get a first impression of the AML, we summarize characteristic language features offered by the AML for modeling distributed embedded systems.

Requirements Classification. One essence of an architectural language is to provide a fixed vocabulary (or ontology) for talking about architectural issues. The AML introduces an ontology which is well suited for systems in the automotive domain. The requirements classification rests upon this ontology. The requirements classification itself is used as a tool to manage the transition from informal, few structured requirements to model aligned, structured requirements. With this classification in mind, the different architectural entities of a system can be identified.

Abstraction Levels define restrictive views on the system model to structure and filter information. Each abstraction level is based upon the more abstract levels. So in a more technical level access to the information contained in a more abstract level is permitted. All of these views show the system on a uniform technical level. At each abstraction level semantic properties are considered which are characteristic for the corresponding technical level. In the development process, the transition from one abstraction level to another abstraction level means to restrict the design space by finding a solution for a specified problem

Formation of Variants. This modeling concept allows to specialize architectural elements according to the context the element is used in. From a methodological point of view the relation between elements and their variants allows to manage complexity by abstracting specialized details to general needed model information. In contrast to the concept of inheritance in object orientation building variants from model elements just means to select specific subelements from the available set of subelements.

Architecture Specific Modeling Concepts. Known ADLs offer recurring modeling concepts for representing certain architectural aspects of a system. Apart from elderly Module Interconnection Languages (MILs), architectural concepts offered by ADLs can be comprised by following equation: "ADL=Components+Ports+Connectors+ Styles" [RS00, BSW94, BDD+93]. In the AML these concepts are applied to different kinds of architectural elements contained in the system model.

Semantic Domain. Each modeling concept of the AML can be represented by various notations. Notations may be textually, tabularly or graphically aligned. Especially the graphical notations, also known as box and line drawings yield profit by the mapping to the AML. Each construct of the notation can be expressed by a corresponding AML modeling concept and therefore inherits its semantics. Within the project Automotive we define mappings of parts of the AML to the UML, to the ASCET-SD [WWWb], and to textual representations. These mappings also define the transformation of models conforming to AML in DOORS, The UML Suite, and ASCET-SD.

3 Towards an Automotive Specific Architecture

Current research in the field of embedded automotive systems reveals the importance of automotive specific concepts in terms of an expert system architecture. As long as commonly accepted abstractions, understood in terms of reusable ontological entities, are not found, we consider domain specificity as desirable. In fact, collaborating in a domain specific manner might well be the only way to identify generally applicable abstractions.

The AML comprises notions which are well known in the automotive domain such as signals, functions, electronic control units, real-time operating systems, communication infrastructure, and processors for assembling the automotive embedded systems architecture. Each of these notions constitutes a fragment of the architectural model at a distinct level of abstraction.

In the sequel we list all notions of this ontology with respect to their classification within a system of AML relevant abstraction levels. We informally describe their semantics and their use as modeling concepts. In addition the AML offers general applicable modeling concepts such as hierarchical structuring, instantiation, formation of variants, formation of configurations and model composition for each presented ontology.

Signals. The abstraction level signals contains model information about the system with the lowest amount of technical details. The core modeling concepts at this stage

are signals and actions. Signals are elementary entities which can be exchanged between actors, sensors, and control units. Each signal can be measured or computed from a physical context. For the construction of architectural models, the model-based management of all signals occurring in a car is essential since their number goes far beyond ten thousand. Furthermore actions allow the modification of signal configurations with respect to a managed set of operations. Both concepts together provide in addition to an ordering mechanism enough modeling power to describe scenarios. Scenarios are ordered sequences of actions which are necessary to achieve a determined goal in a certain context.

Functions constitute basic building blocks at a high level of abstraction independently of later used implementation techniques or target languages. Particularly functions are considered which behave as abstractions of later used control units, actors, sensors, or the environment. Each function is provided with an interface stating the required and the offered signals. Those interfaces are used to model in- and out-ports of functions. For reuseability reasons functions prohibit the access to local signals by putting a scope on them. Therefore communication has to be handled explicitly via signal passing between ports.

One essential model content of architectural models is the explicit representation of signal dependencies between different control functions. Since functions respectively their instances are potentially distributable units that can be deployed to different control units, the consistent and the complete capture of model information in terms of signal dependencies between functions supports the analysis of functional networks and further the collaboration of distributed development teams.

Logical Architecture. The logical systems architecture is determined by the specification of logical partitions where fragments of the functional network are deployed to. These logical partitions characterize potential control units (in AML terminology ”functional clusters”), actors, sensors and the environment. At this stage the uniform treatment of the overall system is broken up to a set of independent subsystems working interactively together.

Comprehensive experience from the development of electronic control units reveals that a clear separation between the logical system architecture level and the technical system architecture level is very helpful when it comes to the partitioning of functions on ECUs. On the logical architecture level only a subset of partitioning criteria is applied in order to achieve a clear view of the functional structure - without identifying the set of functions which constitutes an ECU. However, finally the complete set of partitioning criteria (e.g. also those which consider geometric requirements) has to be applied.

Technical Architecture. On the one hand the technical architecture level is determined by the finalization of the responsibility of each control unit by the application of the full set of partitioning criterias given in terms of technical, economical, quality, and political constraints. On the other hand it is determined by the model-based connection of functions and logical clusters to models of the technical infrastructure (processors, real-time operating systems, and communication infrastructure).

Implementation. At the implementation level the realization of the model in hard- and software is regarded. Altogether this level takes up an exceptional role in the system of abstraction levels since no further information is added to the model. Code generation and the installation of hardware goes far beyond the realization of first prototypes which could be generated from the models gained above. Whereas there are many examples for a successful application of simulation and code generation facilities for testing and for rapid prototyping, code generation often fails to fulfill domain specific constraints. Therefore the generated code has to be manually optimized with respect to code size and execution time. These optimization steps are achieved at this level.

4 Conclusion

Motivated by the aim to meet the challenges of developing complex networks of heavily interacting ECUs, we have developed a system model comprising all necessary constituents for a model-based software development in the automotive domain. In this short paper we have presented major modeling concepts of the AML along with their classification within a system of abstraction levels.

Whereas the AML establishes the basis for an adequate modeling of software in the automotive domain - especially for ECU networks - the system of abstraction levels additionally provides means for structuring the development process according to different domain-specific categories. Our application of these concepts to a common realistic example, a window lifting control system, reveals the benefits of applying our approach.

Future work will cover the following two directions:

First of all, we plan to complete the automotive specific system model: On the one hand this comprises the formal and the complete definition of up to now informally and insufficient described parts. For example consistency dependencies between two adjacent abstraction levels have to be defined in an unambiguous way. On the other hand the concrete development process has to be defined based on the system of abstraction levels by providing rules and heuristics, how to use these levels.

Second, the tool supported transformation from non-executable models to executable, target-dependent code will be explored in order to achieve the long-term objective of a seamless, complete software development process for automotive applications.

5 Acknowledgments

We thank Michael von der Beeck, Jianjun Deng, Ulrich Freund, Bratislav Miric, Bernhard Schätz, and Christian Schröder for helpful discussions and for many comments on draft versions of this paper. We are much obliged to our colleagues of the project Automotive for many fruitful discussions and we thank Manfred Broy for directing this research. This

work has been partially funded by the Bayerische Forschungsstiftung (BayFor) within the Forschungsverbund für Software Engineering II (FORSOFT II).

References

[And99] Jesper Andersson. Die UML echtzeitfähig machen mit der formalen Sprache SDL. *OBJEKTspektrum*, (3), 1999.

[BDD+93] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems, An Introduction to FOCUS - Revised Version. Technical Report TUM-I9202, Technische Univerität München, 1993.

[BR00] Peter Braun and Martin Rappl. Model based Systems Engineering - A Unified Approach using UML. In *Systems Engineering - A Key to Competitive Advantage for All Industries Proceedings of the 2nd European Systems Engineering Conference (EuSEC 2000)*. Herbert Utz Verlag GmbH, 2000.

[BRS00] Peter Braun, Martin Rappl, and Jörg Schäuffele. Softwareentwicklungen für Steuergerätenetzwerke - Eine Methodik für die frühe Phase. In *VDI-Berichte*, number 1547, page 265 ff. VDI, 2000.

[BSW94] Garth Gullekson Bran Selic and Paul T. Ward. *Real-Time Object Oriented Modeling*. John Wiley, 1994.

[CEK+00] T. Clark, A.S. Evans, S. Kent, S. Brodsky, and S. Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. Technical report, pUML Group and IBM, 2000. Available at http://www.puml.org.

[Gro01] U2 Partner Group, editor. *Unified Modeling Language 2.0 Proposal*. U2 Partner Group (http://www.u2-partners.org), 2001. Initial Submission to OMG RFP ad/00-92-02.

[OMG99] OMG, editor. *OMG Unified Modeling Language Specification*. Object Management Group, http://www.omg.org, March 1999. Version 1.3 alpha R5.

[RS00] Bernhard Rumpe and Andy Schürr. UML + ROOM as a Standard ADL? In *Engineering of Complex Computer Systems, ICECCS'99 Proceedings*. IEEE Computer Society, 2000.

[vdBBRS01] Michael von der Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Modellbasierte Softwareentwicklung für automobilspezifische Steuergerätenetzwerke. In *VDI-Berichte*, number 1646, page 293 ff., 2001.

[vdBBRS02] Michael von der Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Automotive Software Development: A Model Based Approach. In *SAE World Congress 2002*. Society of Automotive Engineers, Inc., 2002.

[WWWa] Homepage Automotive (FORSOFT). http://www.forsoft.de/automotive/.

[WWWb] Homepage ETAS GmbH. http://www.etas.de/.

[WWWc] Homepage Telelogic AB. http://www.telelogic.de/.

Towards Service-Based Flexible Production Control Systems and their Modular Modeling and Simulation*

Holger Giese and Ulrich A. Nickel

University of Paderborn
Warburger Straße 100
D-33098 Paderborn
Germany
[hg|duke]@uni-paderborn.de

Abstract: Modeling of modern production plants often requires that the system provides means to cope with frequent changes in topology and equipment and can easily be adapted to new or changing requirements. For validation in form of simulation, however, usually a complete specification of both, the production control software and the physical elements of the manufacturing plant is required. We therefore proposed to use a service-based architectural approach to build the control software using more rigorous separation by means of well-defined interfaces following the software component paradigm. We present an extension of ROOM that further facilitates service-based design and permits the independent validation of components for such a design style. We show how the combination of both concepts permits the compositional validation of the system and thus enables early design validation even for flexible systems. The presented approach further reduces the validation overhead imposed by design evolution as long as local component properties are considered and component interfaces are stable.

1 Introduction

Today's production plants are often characterized by the facility of manufacturing individual goods with small lot sizes. There are many product variants, which means that one has to employ flexible manufacturing systems which can easily be adopted to the new requirements. The available production equipment may also change temporarily due to downtimes and durable when the production capacity has to be adjusted somehow. Current production systems face therefore three major problems: (1) Production control software needs to become decentralized to increase their availability. It is not acceptable, that a failure of a single central production control computer or program causes hours of downtime for the whole production line. (2) Production control software becomes more and more complex. In contrast to the traditional mass production, today's market forces demand smaller lot sizes and a more flexible mixture of different products manufactured in parallel on one production line. (3) The production control software architecture is required to support a flexible extension mechanism to adjust a system rapidly when additional equipment is available or equipment has to be removed.

* This work has been supported by the German National Science Foundation (DFG) grant GA 456/7 ISILEIT as part of the SPP 1064

These problems can be addressed by flexible production control software that employs autonomous operating software agents and a service-based overall software architecture. Some of these production agents will control specific parts of the overall production system like a single manufacturing cell or a transport robot. Other production agents will take the responsibility for manufacturing certain kinds of goods. Such flexible autonomous production agents need knowledge of manufacturing plans for different goods and of their surrounding world, e.g. the layout of the factory or the availability of manufacturing cells. In addition, such production agents have to coordinate their access to assembly lines with other competing agents.

Assume, that a new kind of good shall be produced. If one has to change the topology of a system, often big parts of the control software have to be specified anew. This causes long down times due to extensive tests. An approach would be to simulate the modified specification of the control software, beforehand. Such a direct simulation approach implies, that we have a complete specification of both, the production control software, and the physical elements of the manufacturing plant. In a large production system such changes may happen frequently, which in the worst case would cause an extensive adaption of some parts of the specification. But, for the agent interaction the exact behavior of the agents is not relevant. It is only important, that the agents can communicate correct with each other via a particular protocol. Other details are not relevant and we therefore have to avoid such direct implementation dependencies and apply the traditional software engineering principles of separation and modularization. A first attempt towards the modular design and simulation of production control software has been developed in [GN01]. In this paper this work is extended towards support for more flexibility in form of service-based architectures.

The rest of the paper is organized as follows. In section 2 the used case study and techniques are described. The problems of missing modularization and separation are then discussed in section 3, where the concepts of component-based and service-based design are introduced. Some inherent restrictions of current contract-based specification approaches w.r.t. component behavior and simulation are also discussed. In section 4 we present an approach, that supports the required compositional component notion and partial model simulation. The resulting evaluation scenario in form of simulations is described in section 5 and the paper is finally concluded.

2 Modelling Flexible Production Control Systems

This paper uses the simulation of a simple production process as running example. This production process models a factory with various manufacturing places and with shuttles transporting goods from one manufacturing place to another. The example stems from the ISILEIT project funded by the German National Science Foundation (DFG). The goal of the project is the development of a formal and analyzable specification language for manufacturing processes. This specification language shall allow us to verify important system properties like lifeness and the absence of deadlocks. In addition, a code generator shall provide automatic code generation for the building blocks of a manufacturing process, namely shuttles, gates, storages, assembly lines, etc. The used flexible manufacturing system case study is realized with a track based material flow system, which transports the

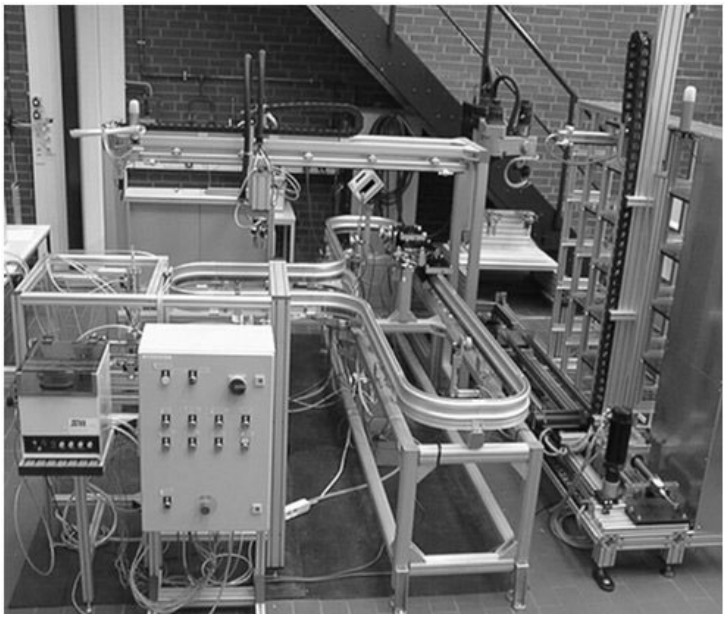


Figure 1 Snapshot of a production system

goods to the different robots or working stations (see Figure 1). Note, that in the current employed case study the physical shuttles are not equipped with a computational device. Therefore, additional external control software on additional computation nodes that handles the shuttles is required.

Figure 2 shows a schematic overview of our case study. On this production line we produce bottle openers, which consist of several components. The system is specified in a way, that one can assign a production task to a shuttle, which means that one shuttle is responsible for the production of certain components. The first step in the working task is to move to station 1, where somebody equips the shuttle with the appropriate material. A display shows the worker, which pieces are needed. By pushing a button, the worker signals the material flow system, that the shuttle is completely equipped. The shuttle now moves to station 2, where the portal robot takes the material from the shuttle and hands it over to the rotator, where the required manufacturing step is performed. After that, the portal robot takes the assembled good from the rotator and puts it on the waiting shuttle. The shuttle now moves to the storage (station 4) where the good is stored. If the control station does not assign a new task to the shuttle, it will rotate on the main loop, until it gets a new task again. Note, that station 3 does not have any functionality at the moment. In the near future, it will connect this production line to a second one.

In our case study, we use Programmable Logic Controllers (PLC) to control the stations of the production line. For the specification of the behavior of such a controller, different approaches exist, which cover different abstraction levels. Sequential Function Charts for example describe the sequence of a PLC program as a state transition diagram, whereas Structural Text (ST) is a notation similar to PASCAL. The PLC programming standard

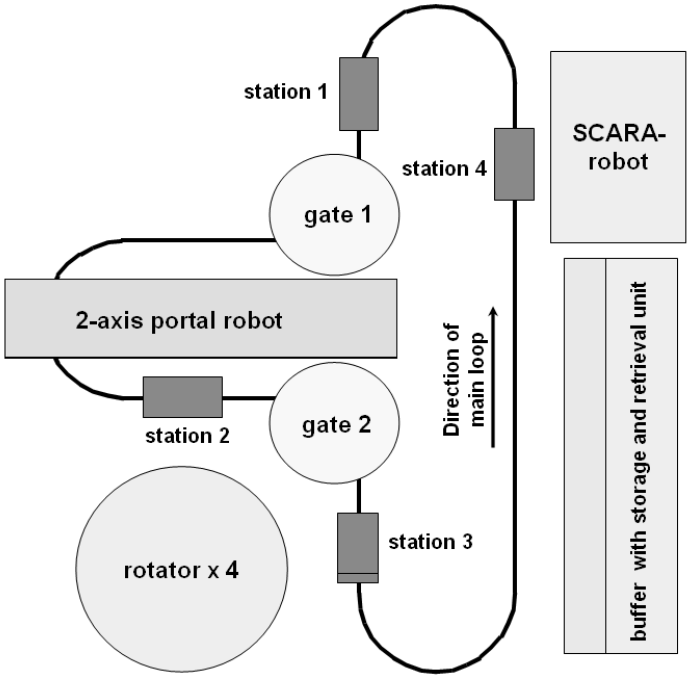


Figure 2 Topology of our sample factory example

IEC61131-3 [Int93] tries to integrate these languages based on a common concept of data types, variables and program organization units.

The behavior of embedded system processes is also often specified either using SDL process diagrams [ITU99] or using statecharts [HG96]. Both notations basically model finite state automata which react on signals by executing some actions, sending signals, and changing to new states. Both languages have a well defined formal semantics and tool support for analysis and simulation and code generation is available [Dou98, AT98, RoR].

However, the discussed languages do not provide appropriate means for the specification of complex application specific object-structures as required by the described autonomous production agents.

Common object-oriented modeling languages, like UML, support the modeling of complex application specific object-structures. UML focuses on early phases of the software life-cycle like object-oriented analysis and object-oriented design, cf. [BRJ99]. Thus, UML behavior diagrams, like collaboration and sequence diagrams, usually model typical scenarios describing the desired functionality. Only UML state diagrams - an adjusted version of the original statecharts [HG96] - provides means to model reactive behavior. For the real-time and embedded system domain, ROOM [SGW94] and its successor UML-RT [SR98] provide the required more domain specific adjustments. See [RSRS99] for a proposal how to map the concepts of the ROOM approach to the UML.

3 Towards Flexible and Modular Design and Evaluation

Assume, that a new kind of good shall be produced. If one has to change the topology of the material flow system, e.g., often big parts of the control software have to be specified anew. This causes long down times due to extensive tests. Thus, in [NN01] we describe an approach of how we can simulate the modified specification of the control software, beforehand. The Fujaba CASE tool [Fujaba] can be used to generate executable Java code and to observe its execution using the Java reflection library. Production sequences can be visualized and analyzed. The simulation is based on a simulation kernel, which serves as a model for the physical components of the production system.

Such a simulation approach implies, that we have a complete specification of both, the production control software, and the physical elements of the manufacturing plant. Imagine, that we just change the set-up for the assembly line or the CNC-code of a robot. In a large production system this happens frequently. In the worst case, this would cause an extensive adaption of some parts of the specification. But, for material flow purposes, many details of the behavior of an assembly line are not relevant. It is only important, that a shuttle can communicate with the assembly line via a particular protocol or how long the assembly line takes to perform the next production step. Other details as specified in the statechart of the AssemblyLine class in Figure 3 are not relevant. We therefore have to avoid such implementation dependencies and apply the traditional software engineering principles of separation and modularization.

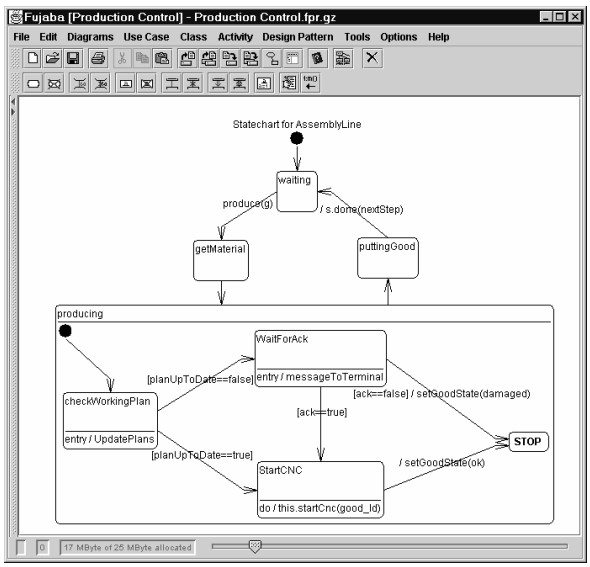


Figure 3 Statechart of AssemblyLine

The description of external phenomena like the AssemblyLine in form of a class is a problematic solution. Traditional class-based object-oriented design does not emphasize separation and thus more rigorous separation by means of interfaces is required when direct class dependencies should be avoided. This implicit treatment does not support the inde-

pendent deployment and composition of parts. The component paradigm [Szy98] therefore demands to consider the contractual relations more explicit to support the systematic exchange of parts. Instead of direct class relations, explicit contracts have to be used.

The concept of evaluation by simulation does also not scale up to complex systems, because the cognitive capacity of a human to keep track with the simulation results visualized by the tool is rather limited in practice. Thus, an overall system simulation is not directly applicable for evaluation of complex systems. A *compositional* component-based and systematic approach for modeling and checking properties is therefore needed, which permits to check system properties in a modular fashion within the original scenario of simulation-based early design evolution.

3.1 Component-Based Design

Building systems by using components is a well known concept from classical engineering disciplines. The reuse of such pre-defined and tested software components allows the engineer to construct systems on a very high abstraction level. This decreases time-to-market and improves the productivity. Thus, effort has been put on adopting these principles to software engineering. A successful example for the application of the component paradigm in the field of industrial process control can be found in [CL00].

In terms of software, a common notion for components does rarely exist [Cou99]. However, most definitions put emphasis on the independence between production and deployment, the composition by third-parties and the explicit specification of context dependencies in a contractual style. Szyperski [Szy98] defines a component as follows:

> A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Following this definition, component technology adopts the principles of object-orientation [RBP+91,Boo93,JCJO96,CAB+94] from the implementation in a programming language environment up to the run-time environment of the system. An application or system is decomposed into runtime elements, that can be build, analyzed, tested and maintained, independently. As Meyer and Mingins stated in [MM99] component technology is based on the solid ground of object technology in applying the encapsulation principle of object-orientation.

For the real-time and embedded system domain, ROOM [SGW94] and its successor UML-RT [SR98] propose a component concept (capsules) with explicit connections points (ports). A very general notion of a protocol is employed to specify general multi-party interactions. This includes binary protocols which are by far the most common ones. The ROOM port concept permits to restrict the possible interaction via a specific connection to a certain set of signals specified by a given protocol role in form of a statechart.

Besides the ports of the ROOM approach, in the literature a number of different notions for contracts have been proposed which can be classified by the following hierarchy [BJW99]: syntactical interface, behavior, synchronization and quality of service. While contracts in form of syntactical interfaces are supported by all typed programming languages and middleware platforms, full behavioral contracts, e.g. in form of pre- and post-conditions, are rarely used in practice. Synchronization contracts can describe the non uni-

form service availability, while also request scheduling solutions such as specific reader/ writer policies are possible. Quality of service contracts further allow to describe the contract behavior w.r.t. time and throughput characteristics, but their platform dependent nature renders their consideration during design and simulation a complex task.

For the production control domain a more detailed notion for contracts than commonly used in programming languages is required. Therefore, a contract has to be a kind of boundary object that extends the UML-RT port concept rather than being simply a pure abstract concept such as an interface. We further restrict the rather general notion of protocols and protocol roles in ROOM to the binary case to exclude the complexity of multi-party interactions and their sheduling [JS96].

In Figure 4 a Factory subsystem developed in this manner is presented. The description employs the UML-RT concepts to denote the provided and used boundary contracts of the factory subsystem in form of ports. As a visual short-cut we use a black box for a main protocol role and a white one for its counterpart. The explicit handling of the main protocol role and its complementary role is also simplified using a single <<contract>> stereotype to specify exactly one main protocol for the providing side and derive the usage protocol of the client side, implicitly.

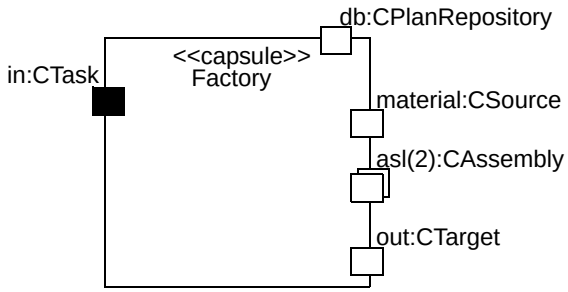


Figure 4 The Factory as component

In Figure 5 for the AssemblyLine class such an explicit contract CAssembly for the assembly line is presented. The whole subsystem of the factory example of Figure 1 can be redesigned in this manner employing the component and contract concepts. In the given factory example the task processing provided via the CTask contract is realized using the CSource, CAssembly, CTarget and CPlanRepository contract.

Following the component paradigm, instead of implicit abstract classes such as AssemblyLine, a contract CAssembly with a protocol is used to describe the component boundary (cf. Figure 5). Thus, some of the problems that arise when the detailed behavior of a component changes, can be avoided. The provided and used contracts of a component are an implicit description of all possible environments and thus result in a well-defined and *complete* test frame. In contrast, in the case of an explicit given class, only *one* specific test scenario is given which consists of the given surrounding classes and their current implementation. The abstraction and decoupling realized via the contracts can be further used to support the modular simulation.

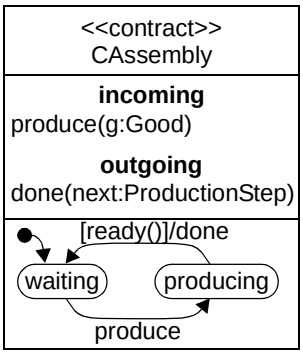


Figure 5 Contract of AssemblyLine

3.2 Service-Based Architectures

As one of the major problems of current production systems, we already identified the need for a more flexible architecture concerning the adjustments due to changes in the system setup - even at run-time. On the software level the dynamic composition of components via services has recently received considerable attention in form of open service-oriented software architectures and web-services (cf. [NET00, ONE01]). Here, each application determines its context embedding by means of dynamic service lookup. This architectural style facilitates the integration of independently developed systems, using service contracts which include meta-data to guide their composition with third-party components in a plug & play manner at run-time.

However, service-oriented software architectures are not really a new concept. The basic principles have been standardized, e.g. in tina-c [CM95] in the telecommunication field or the ISO Open Distributed Processing (ODP) model [ISO95]. Well established middleware approaches such as DCOM [Cha96] or CORBA [Vin98] also support trader and name services for dynamic lookup of services. Newer approaches such as Jini [Sun00], Microsoft .NET [NET00] or Sun ONE [ONE01] further emphasize the service-oriented composition at run-time.

In the domain of production control systems the most prominent approach is OLE for Process Control (OPC) [IL02]. It extends the PC based Microsoft concepts towards the industry and automation domain providing a server browser for remote lookup. Sun's Jini [Sun00] also address embedded devices. However, the main focus lies on small independent devices and the support of ad hoc networking.

To model the service-oriented aspects of a production control software, we have further extended the UML-RT approach. We use solid contract/port connections to describe fixed cooperations while service-based dynamic ones are visualized using a dashed line.

In Figure 6, the internal design of the factory software component is described using both, fixed and dynamic cooperations. Inside the factory control software, a number of shuttle agents are used to describe the autonomous processing required for each requested task. The underlying hardware restrictions (the shuttles itself have no programmable processing

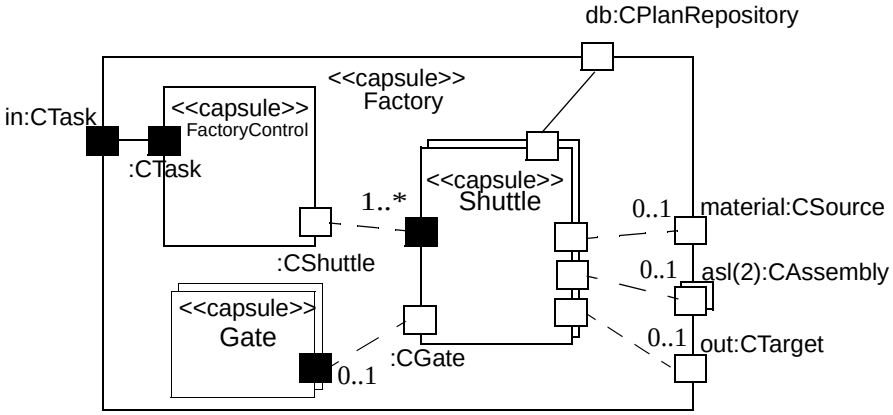


Figure 6 The Factory component internals

units) further enforce a mapping where the shuttle agents virtually control their physical shuttle by communicating with the connected gates to achieve their goals. The required processing further enforces, that the shuttle agent cooperates temporarily with the material depot, assembly lines or output depot using the CSource, CTarget and CAssembly contracts.

The contract CPlanRepository is used only in occasional periods to optimize the performance of the shuttle agents by providing actual informations concerning changes in the floorplan of the factory. However, the autonomy of the agents ensures, that they will adjust to directly observed changes.

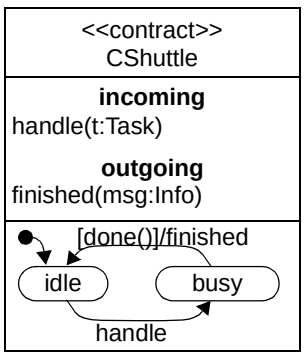


Figure 7 Shuttle contract

Figure 7 depicts the contract used to decouple the overall FactoryControl and the different shuttles. Its basic protocol describes how a task t to process is hand over and autonomously processed by the shuttle control software agent.

In contrast to the common static cooperation scenario in embedded systems the service-based architecture is employed here. Therefore, when the overall factory software control component FactoryControl receives a new request to process a task it dynamically looks

up an available shuttle agent to process the task. Thus, when a shuttle is temporarily not available and therefore no longer registered in the service lookup, the overall control will automatically chose another shuttle, if available.

This is achieved using the contract types to identify reasonable matches between service providers (Shuttles) and service customers (FactoryControl). On the first sight, type equivalence seems to be a reasonable choice for matching. However, the further evolution of systems will then enforce major redesign activities when, for example, extended versions of a shuttle or shuttle software are employed. A more practical choice is inheritance-based subtyping as supported by most programming languages and middleware approaches. This choice assumes, that all further developed software artifacts are using the same unique class and contract hierarchies. If these assumptions are not fulfilled, an alternative approach are efficient notions for contract matching (subtyping) that exploit the type structure or meta-data (cf. [NET00, Sun00, Gie01a]). In this case, we propose to employ notion proposed in [Gie01a], while depending on the supported contract types other notions are also applicable.

4 Behavior and Composition

Due to the fact, that a component is subject to composition by third parties, it has to provide a clear definition of how it can be used. In the object-oriented approach, interfaces or abstract classes are employed to separate usage and implementation concerning the syntactical typing. For component-oriented programming and distributed systems the independent construction of each component further requires that a suitable service contract covering also semantic issues is provided. Otherwise, the required implementation-independent information for the composition by third-parties is not available.

Component systems differ from traditional software products, where a view restricted to the white-box composition and the sequential case fits most often. Instead, the third-party composition of components has to consider black-box composition and often takes place in a concurrent environment. Therefore, composition can result in reasonable synchronization problems, such as deadlocks, that cannot be avoided using specific implementation styles. To address this problem, such composed systems have to be tested or simulated. However, to get a reasonable result, usually a complete specification of the system is required.

Using the protocols of provided and used contracts, even the simulation of an incomplete specification becomes feasible. In a first step, the contract protocols can be used to build the most general possible component environment by generating arbitrary request sequences as guaranteed via the provided contracts and assuming the behavior guaranteed by the used contracts. A simulation can, however, cover only the possible system behavior, but fails for lifeness aspects. When classes represent the component border in an implicit manner, the model simulation can assume progress for each single statechart, because it describes a realization which is executed. In contrast, to assume progress for a given set of provided and used contracts will also result in the obligation of connected components to serve them in an independent manner. In practice, however, provided and used behavior do often depend on each other and therefore progress and lifeness properties can not be handled for each contract in isolation.

Another crucial question w.r.t. progress and the provided component contracts is, whether different clients are served in a fair manner. If no fairness is assumed, a set of client requests will only exclude the blocking of one of its members, when the direct or indirect synchronization of the clients itself exclude that one client can rule out any other one. Therefore, to implement fairness in an explicit manner based on a set of unfair operating components is a rather hopeless undertaking. The single components should instead guarantee to process the different client request in a fair manner, e.g., using a fair request queue.

The described contract concept as short-cut of the port and capsule extension thus permits to evaluate whether the component connections are used and provided in a protocol conform manner. However, the proposed protocols and contracts are not sufficient to achieve the intended modular form of designs which supports validation by simulation. The protocol restrictions do not address lifeness properties and therefore fail to describe the component environment as required.

To overcome this limitations for contract protocols, we extend the used notion of statecharts and further distinguish *progress* and *quiescent* transitions [Rei98] denoted by usual and dashed arcs, respectively. We further demand that progress is guaranteed by the contract provider and all possible provider events are never blocked by the clients. Therefore, a secure usage will only result in a situation where clients will wait for the answers or guaranteed state changes of the provider, whereas the provider can never be blocked by a client.

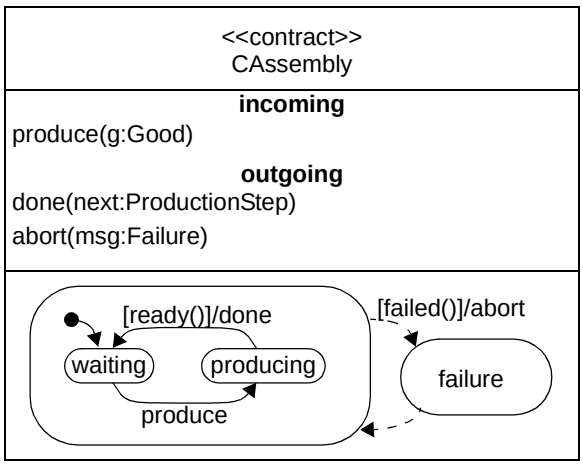


Figure 8 Addjusted CAssembly contract

In Figure 8 quiescent transitions from the states waiting and producing leading back and forth to a state failure are used to also take the possible malfunction into account. These transitions might occur, but may also never occur. In contrast for the other progress transitions hold that one transition will occur if at least one is enabled. Note, that no history state is used and therefore the semantics of the statechart requires that in case of a failure the loaded material is always removed manually before the shuttles become operational again.

The progress guarantee for provided contracts results in arbitrary protocol conform usage by the test environment. For the used contracts the guaranteed progress is employed during

simulation and therefore only relevant cases of permanent blocking are observed. The boundary contracts CTask, CSource, CTarget, CAssembly and CPlanRepository have to be combined with the executable model of the factory example to obtain the intended test scenario. A task request may occur and the appropriate processing by the shuttles is initiated. The used contracts CSource, CTarget, CAssembly and CPlanRepository are requested as specified exploiting the progress property. The progress of a contract, however, cannot always be guaranteed by realizing the component independently of the component environment and the interplay with other components. The described test environment construction is therefore only valid in a strictly layered architecture.

If more general forms of architectures are considered the relation between provided and used contracts of components cannot be ignored. In contrast to safety properties it is problematic to ensure lifeness properties such as progress for arbitrary connected components. Therefore, commonly the whole external relevant component behavior described in form of processes such as formalized by CSP [Hoa85], CCS [Mil89] or LOTOS [ISO89] have to be considered.

An overwhelming variety of preorders and congruences for process refinement and abstraction have been proposed for these process algebraic approaches to address whether a given external component specification is realized correctly. The proposed relations, however, have to ensure substitutability [LW93] w.r.t. any possible process environment and therefore often result in very tight specification realization relations. The relations enforce that the specification has to reveal too many details of a realization, e.g., the complete buffering effects, and therefore do not provide the necessary degree of separation.

In a layered architecture with acyclic module usage relations, in contrast, a separate treatment of lifeness properties becomes feasible [LS94] by exploiting the acyclic nature of dependencies. We generalize this idea to support separation for progress properties even for non-layered structures employing explicit contract dependencies for a given set of provided and used contracts.

In [Gie00,Gie01b] even the combination with explicit partial external specifications covering a subset of the used or provided contracts of a component are presented. To specify such complex dependencies between multiple provided and used contracts so-called *complex contracts* describing the explicit behavior and synchronization can be employed. By including the traditional case of a complete external specification in form of a specification process the approach supports a whole spectrum of possible component descriptions varying w.r.t. the degree of abstraction and embedding restrictions.

For the factory example we consider only the simplest case to build the needed overall component behavior based on explicit specified contract dependencies. If a used contract is required to realize the behavior of a provided contract, we have to declare a dependency between them to make this regress explicit. We further have to restrict that for the composition of components the concatenated dependency relations remains acyclic. The specified dependencies therefore guide the possible component composition by demanding that a component embedding never results in a cyclic progress dependency [Gie01b].

The construction of the dependency relation for the elements of the Factory component can be easily determined considering which used contracts are required to ensure progress of the provided contract. Following this simple scheme in Figure 9 we can conclude that

the CTask contract of the FactoryControl requires the Shuttle contracts and thus depend on them. For the Shuttle holds that its provided behavior requires the CSource, CAssembly, CTarget and CGate contracts, while the CPlanRepository contract is required only sometimes to update the agent information concerning the floorplan of the factory. If it s temporarily not available this will not hinder the agent to fulfill its task. Therefore, no dependency has to be declared.

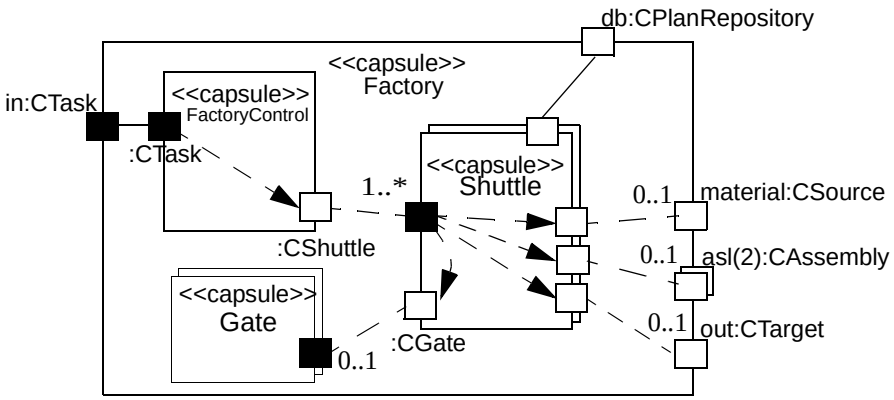


Figure 9 The Factory component internal dependencies

In Figure 10 the dependencies between the provided CTask contract of the Factory component and the used contracts are specified. The construction of the dependency relation for the Factory component can be easily derived considering its internal design as depicted in Figure 9 using the following simple rule: The dependencies of a component are given by the concatenation of all internal port connections and the dependencies of the contained components. Thus the cooperation of the CSource, CTarget and CAssembly contracts are definitely required for providing the Task contract, while the CPlanRepository contract is only used in a sporadic manner to update the locally stored machine control programs.

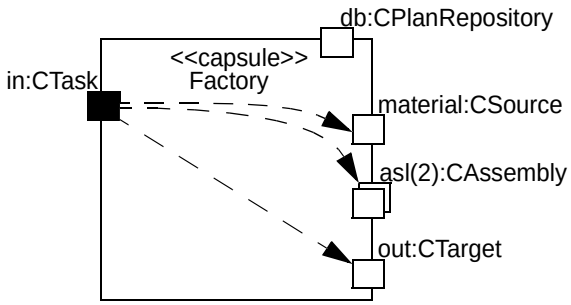


Figure 10 The Factory component and its dependencies

5 Modular Evaluation

Current approaches to evaluate software systems and their architecture which take concurrency problems into account [MDEK95, LAK+95, GKC99] do not address an open dynamically composed component system that permits the plug & play for independent developed components.

The proposed approach, however, can evaluate the software components in a suitable test environment by using the additional information provided by the contract protocols and dependency relation. While the provided and used contract protocols are combined as described before, the progress of provided and used contracts of the test environment is non-deterministically controlled as specified by the contract dependency relation. When the test environment in a specific state is waiting for progress at the CTask contract, it has to provide progress for the CSource, CAssembly and CTarget contracts. The CPlanRepository contract, in contrast, needs not to be served in this state

Besides the component internal dependencies between provided and used contracts, the overall component behavior of a factory is of interest. We propose to use a set of UML sequence diagrams to specify the necessary component behavior. Each time the prefix of one such specified trace is present, we further demand that the overall partial trace has to be conform with the one specified by the sequence diagrams. Otherwise, the realization of the component does not conform to that sequence diagram.

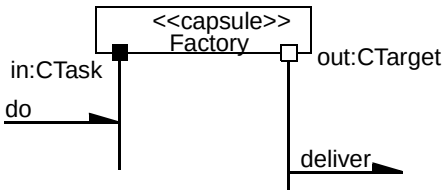


Figure 11 Requirement of guaranteed product delivery

In Figure 11 the expected property for a single do request to a factory is specified. Using a sequence diagram and the scenario based techniques permit to consider this requirement in separation. The factory realization will process multiple do request in parallel, but for each single request the presented sequence diagram can be used as necessary behavioral property. A simulation scenario should therefore take track of the initiated do requests and whether the specified related deliver requests have occurred.

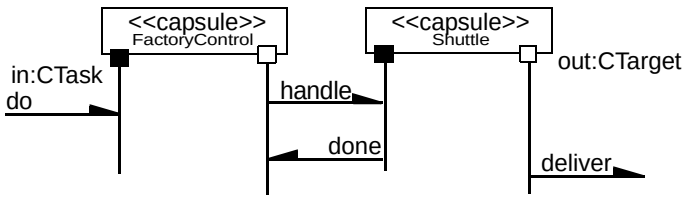


Figure 12 Decomposed requirement of guaranteed product delivery

In Figure 12 the decomposition of the required property for a single do request to the elements of the factory is depicted. Splitting the a sequence diagram a test scenario for the two relevant components FactoryControl and Shuttle is derived.

6 Conclusion and Future Work

In this paper a concept for the design of flexible production control system software that enable partial model evaluation in form of simulation has been presented. It has been discussed which extensions to traditional contract-based component separation are required to end up in a useful and appropriate service- and component-based architecture which supports partial simulation. The proposed technique to describe the component behavior and component environment does further provide the necessary restrictions to exclude unexpected component embeddings which invalidate the assumed dependencies for the provided and used contracts of a component. We plan to further extend the presented ideas to also address besides simulation model checking [CGP00]. Also contracts and requirements should include quality of service aspects such as worst case execution times or throughput guarantees. Also more elaborated tool support for the design and evaluation of complex real-time systems with this approach is planned.

References

[AT98] J. Ali and J. Tanaka. *Implementation of the Dynamic Behaviour of Object Oriented Systems*. In Proc. of the 3rd Biennal World Conference on Integrated Design and Process Technology, pages 281–288. ISSN No. 1090-9389, Society for Design and Process Science, 1998.

[BJW99] Antoine Beugnard, Jean-Marc Jezequel, and Damien Watkins. *Making Components Contract Aware*. IEEE Computer, 32(7):38–45, July 1999.

[Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Menlo Park CA, 1993. (Second Edition).

[CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[Cha96] D. Chappell. *Understanding ActiveX and OLE - A Guide for Developers and Managers*. Microsoft Press, 1996.

[CL00] Ivica Crnkovic and Magnus Larsson. *A case study: demands on component-based development*. In Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, 2000.

[CM95] Martin Chapman and Stefano Montesi. *Overall Concepts and Principles of TINA, Version: 1.0*, February 1995.

[Cou99] William T. Councill. *Third-Party Testing and the Quality of Software Components*. IEEE Software, 16(4):55–57, 1999.

[Dou98] B.P. Douglass, editor. *Real Time UML*. Addison-Wesley, 1998.

[Fujaba] Software Engineering Group, University of Paderborn. *Fujaba: From UML to Java and Back again*. http://www.fujaba.de/.

[Gie00] Holger Giese. *Contract-based Component System Design*. In Jr. Ralph H. Sprague, editor, Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33), Maui, Hawaii, USA. IEEE Press, January 2000.

[Gie01a] Holger Giese. *Object-Oriented Design and Architecture of Distributed Systems*. Phdthesis, Westfälische Wilhelms-Universität Münster, Fachbereich Mathematik und Informatik, February 2001.

[Gie01b] Holger Giese. *Typed Component Systems, Version 1.0*. TechReport Bericht tr-ri-01-224 Reihe Informatik, Fachbereich Mathematik-Informatik, Universität Paderborn, May 2001.

[GKC99] D. Giannakopoulou, J. Kramer, and S.C. Cheung. *Analysing the Behaviour of Distributed Systems using Tracta*. Journal of Automated Software Engineering, special issue on Automated Analysis of Software, 6(1):7–35, January 1999.

[GN01] Holger Giese and Ulirch A. Nickel. *Towards Modular Modeling and Simulation of Production Control Systems*. In Object-oriented Modeling of Embedded RT-Systems (OMER-2), Herrsching am Ammersee, Germany, May 2001.

[HG96] D. Harel and E. Gery. *Executable Object Modeling with Statecharts*. In Proc. of the 18th International Conference on Software Engineering, Berlin, Germany, pages 246–257. IEEE Computer Society Press, May 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[IL02] Frank Iwanitz and Jürgen Lange. *OLE for Process Control. Fundamentals, Implementation and Application*. Decker/Müller, 2002.

[Int93] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers - Programming Languages, IEC 61131-3*, 1993.

[ISO95] ISO/IEC. *Open Distributed Processing Reference Model - parts 1,2,3,4*, 1995. ISO 10746-1,2,3,4 or ITU-T X.901,2,3,4.

[ITU99] ITU - Telecommunication Standardization Sector. *Z.100: LANGUAGES FOR TELECOMMUNICATIONS APPLICATIONS - SPECIFICATION AND DESCRIPTION LANGUAGE*, November 1999.

[JCJO96] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1996. (Revised Printing).

[JS96] Yuh-Jzer Joung and Scott A. Smolka. *A comprehensive study of the complexity of multiparty interaction*. Journal of the ACM, 43(1):75–115, January 1996.

[LAK+95] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Vera, D. Bryan, and W. Mann. *Specification and analysis of system architecture using Rapide*. IEEE Transactions on Software Engineering, 21(4):336–355, April 1995.

[LS94] S. S. Lam and A. U. Shankar. *A theory of interfaces and modules i-Composition theorem*. IEEE Transactions on Software Engineering, 20(1):336–355, January 1994.

[LW93] Barbara Liskov and Jeannette M. Wing. *Specifications and Their Use in Defining Subtypes*. In Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA93), pages 16–28, 1993.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. *Specifying distributed software architectures*. In Fifth European Software Engineering Conference (ESEC'95), volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Verlag, September 1995.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[MM99] Bertrand Meyer and Christine Mingins. *Component-based Development: From Buzz to Spark*. IEEE Computer, 32(7):35–37, July 1999.

[NET00] *Microsoft .NET: Realizing the Next Generation Internet*. Microsoft, June 2000. White Paper.

[NN01] U.A. Nickel and J. Niere. *Modelling and Simulation of a Material Flow System*. In Proc. of Workshop 'Modellierung' (Mod), Bad Lippspringe, Germany. Gesellschaft für Informatik, 2001.

[ONE01] Sun[tm] Open Net Environment (sun ONE) Software Architecture. Sun Microsystems, Inc., 2001.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms, Modeling and Analysis with Petri Nets*. Springer Verlag, 1998.

[RoR] Rational. *RR-RT, the Rational Rose Real Time case-tool. Online at http://www.rational.com*.

[RSRS99] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. *UML + ROOM as a Standard ADL?*. In F. Titsworth, editor, Engineering of Complex Computer Systems, ICECCS'99 Proceedings. IEEE Computer Society, 1999.

[SGW94] Bran Selic, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[SR98] Bran Selic and Jim Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. Techreport, ObjectTime Limited, 1998.

[Sun00] Sun Microsystems. *Jini Specification*, October 2000. Revision 1.1.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Vin98] Steve Vinoski. *New Features for CORBA 3.0*. Communications of the ACM, 41(10), October 1998.

Implementing Function Block Adapters

Torsten Heverhagen, Rudolf Tracht
University of Essen, Germany
FB 12, Automation and Control
Torsten.Heverhagen@uni-essen.de, Rudolf.Tracht@uni-essen.de

Abstract: Function Block Adapters (FBAs) are new modeling elements, responsible for the connection of UML capsules and function blocks of the IEC 61131-3 standard. FBAs contain an interface to capsules as well as to function blocks and a description of the mapping between these interfaces. In this paper we discuss implementation issues of FBAs. While the specification of FBAs is completely platform-independent, we show that different hardware solutions force highly platform-dependent implementation models. In most cases a FBA is implemented in two programming languages - an object oriented language and a language out of IEC 61131-3. While object oriented programs mostly implement an event-driven execution semantic, PLC programs are executed cyclically. Especially this heterogeneous implementation environment was the motivation for developing Function Block Adapters.

1. Introduction and Motivation

Programmable Logic Controllers (PLCs) are widely used for controlling industrial manufacturing systems. The programming of PLCs is normally done in special languages defined in the IEC 61131-3 standard [2]. The increasing complexity of the controlling software for manufacturing systems leads to the need for more powerful specification languages. Latest developments in object oriented technology like UML-RT (successor of ROOM [4]) face this need [1]. But in most cases it is not possible to substitute PLCs in existing plants completely with object oriented systems. Therefore, our approach is to integrate object oriented technology (UML-RT) into an existing PLC-environment in the case of extending a manufacturing system with new components without throwing away the PLC. New components can be for example an Industrial Personal Computer (IPC) which is connected over a fieldbus system to the PLC. We assume, that the IPC program is then designed with UML-RT.
In [5], [7] we introduced a new UML stereotype, the Function Block Adapter (FBA), which is responsible for the connection of UML-RT capsules and function blocks of the IEC 61131-3 standard. FBAs contain an interface to capsules as well as to function blocks and a description of the mapping between these interfaces. For this description a special FBA-language is provided. The FBA-language is easy to understand both to UML-RT and to IEC 61131-3 developers, so they can unambiguously express the interface mapping. An important advantage of the FBA-language is the possibility to use it at an early design state of the UML-RT system.

In this paper we discuss implementation issues of FBAs. We show that different hardware solutions force highly hardware-dependent implementation models. In most cases a FBA is implemented in two programming languages – an object oriented and an IEC 61131-3 programming language. While object oriented programs mostly implement an event-driven execution semantic, PLC programs are executed cyclically. A closer look into cyclic program execution is given in section 2.2. Especially this heterogeneous implementation environment was the motivation for developing Function Block Adapters.
First some example requirements are given in section 2. Section 0 gives a short introduction into Function Block Adapters by an example based on section 2. In section 4 two possible hardware solutions are discussed in principle. As an intermediate step towards implementation section 5 introduces an abstract execution model the example FBA. A hardware solution with a fieldbus of type Profibus-DP and a PLC of type S7-300 is discussed in section 6. Section 7 closes this paper with a summary and outlook.

2. Example Requirements

Assuming that there is a PLC on which runs a function block called *MyFB* like shown in Figure 2. It contains three input and three output variables. The Boolean variables *B, C, E*, and *F* are used to provide trigger-events to and from the function block. The data given in *A* is interpreted by *MyFB* depending on the value of *B*. *MyFB* only provides valid data in *D*, when *E* is *true*. Section 2.2 discusses this protocol in more detail.

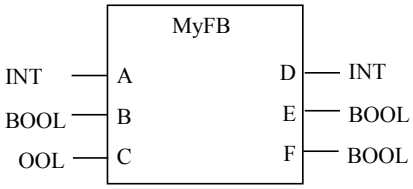


Figure 2. FB interface

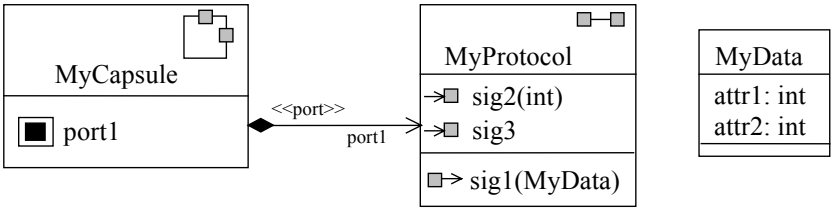


Figure 1. Capsule interface

Furthermore we assume that there is a new application being developed which is designed in UML-RT. This application contains a capsule called *MyCapsule* like shown in Figure 1. *MyCapsule* has to send and receive the signals of protocol *MyProtocol* to and from the function block *MyFB*. This protocol is implemented in *port1*. The mapping from *port1* to the interface of *MyFB* will be explained in section 0.

2.1 The Protocol *MyProtocol*

Figure 3 describes the protocol by a statechart. Initially the protocol is in state *Sig1_or_sig2* if no signal is being transmitted. Transmission of *sig1* is expressed by a transition from state *Sig1_or_sig2* to state *Sig1_or_sig2*. After sending of *sig2* the protocol is in state *Wait_for_sig3*. In this state only sig3 is being able to sent.

State *Sig1_or_sig2* is left by two transitions. If the signals for both transitions arrive at the same time only one transition fires. The selection algorithm is priority-based. In this example the priority of *sig1* is higher than the priority of *sig2*.

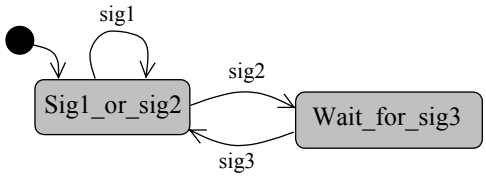


Figure 3. Protocol state machine for *MyProtocol*

2.2 The Protocol of Function Block MyFB

The execution semantic of function blocks is different from the one of capsules. Normally they are executed cyclically. Figure 4 illustrates how program elements are executed within a PLC. After initialization a cycle is started which consists of reading of PLC inputs, program execution, and updating of PLC outputs. The cycle time is mainly determined by the program execution time. Programming languages of IEC 61131-3 contain no statements like waiting for events. If a PLC-programmer implements a waiting or endless loop, the PLC operating system recognizes the loop and refuses the execution. This behavior forces a PLC programmer to a special kind of programming style like explained in section 6.

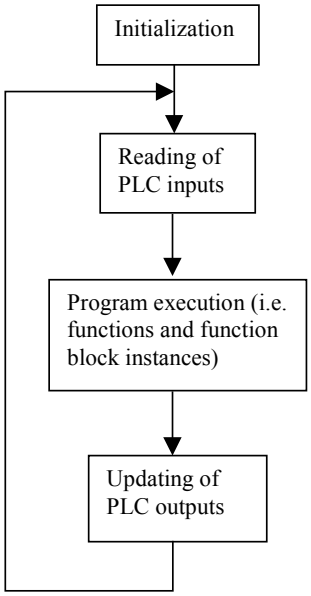

Initialization
Reading of PLC inputs
Program execution (i.e. functions and function block instances)
Updating of PLC outputs

Figure 4

Instead of having in mind the cyclic execution of *MyFB* it is easier to describe its behavior by a timing diagram given in Figure 5. The timing diagram is divided into three sections by dashed lines. The first section shows how *MyFB* reads sequentially two values from *A*. In this paper we call this FB-Signal *B*, because *B* triggers *MyFB* to read a value from *A*. With a raising edge in *F MyFB* acknowledges that *A* was read. This shall correspond to

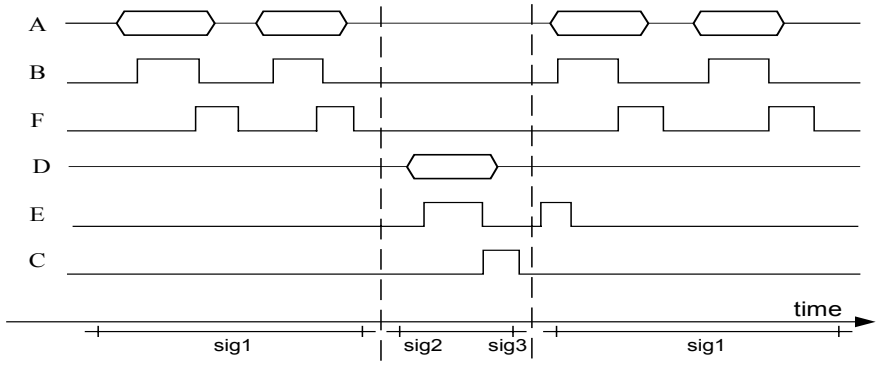


Figure 5

sig1 of *MyProtocol*. The second section describes how MyFB provides some data given in *D* for another function block. We call this FB-Signal *E*, because *E* is the trigger-variable for other function blocks to read a value from *D*. *MyFB* awaits a raising edge in *C* to get an acknowledgement. This shall correspond to the combination of *sig2* and *sig3*

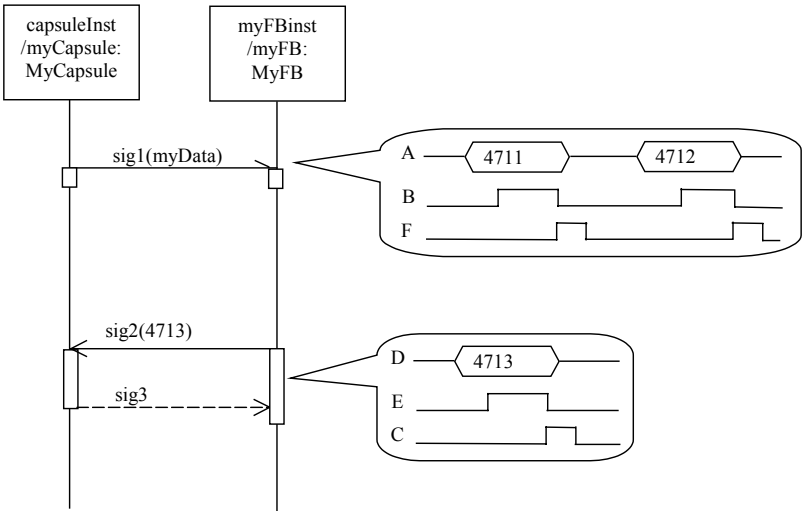


Figure 6. Example interaction

of *MyProtocol*. Section three of Figure 5 shows that FB-Signal *B* has higher priority than FB-Signal *E*. FB-Signal *B* can interrupt FB-Signal *E* only until the raising edge of *C* is being reached.

3. Function Block Adapters

Figure 6 shows a possible interaction between an instance of *MyCapsule* called *capsuleInst* and an instance of *MyFB* called *myFBInst*. At first the signal called *sig1* is sent from *capsuleInst* to *myFBInst*. *Sig1* contains an instance of the data class *MyData*: myData.attr1 = 4711; myData.attr2 = 4712.

Of course the function block *MyFB* cannot receive the UML-Signal without a translation into a FB-Signal. The legend which is attached to *sig1* in Figure 6 shows the assignments of the function block variables, which are needed to give the information of UML-Signal *sig1* into the function block *MyFB*. *MyFB* reads the values of *attr1* and *attr2* as a sequence in the input variable *A*. Input variable *B* is used to signal *MyFB* that valid data is assigned to variable *A*. With the output variable *F MyFB* acknowledges the inputs of variables *A* and *B*.

The second signal *sig2* is sent synchronous. This means that the sender (*myFBinst*) waits for an acknowledgement (*sig3*). Graphically an asynchronous message is displayed by a single sided arrow and a synchronous message by a double sided arrow. The answer to a synchronous message is denoted by a dashed arrow.

The data of *sig2* is given in the output variable *D* of *MyFB*. With output variable *E MyFB* tells that the content of *D* is valid. In input variable *C MyFB* awaits the acknowledgement.

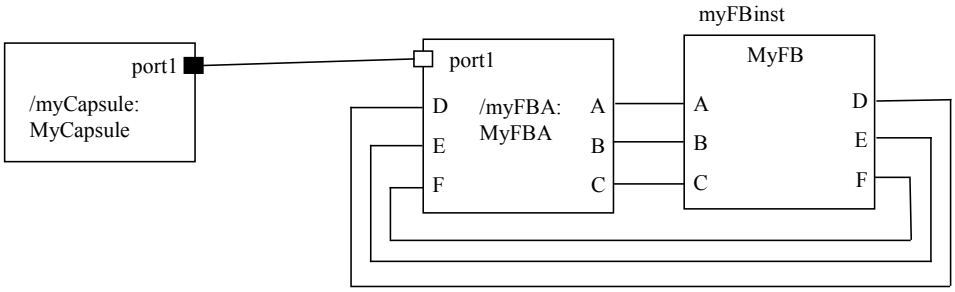


Figure 8. Extended structure diagram for MyFBA

The translation of the timing diagrams into UML-signals is done within Function Block Adapters. Sections 3.1 and 3.2 explain the structure and the behavior of the FBA called *MyFBA*.

3.1 Structure

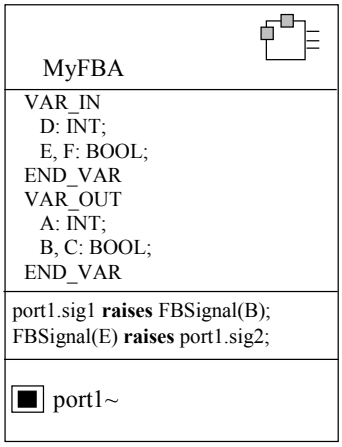


Figure 7. Class symbol for MyFBA

A FBA is a stereotype of a UML class which contains all properties of a capsule. A FBA uses ports to establish connections to other capsules.
Additionally FBAs define interface variables for the communication with function blocks. A FBA can graphically displayed in an extended structure diagram (Figure 8). The FBA *MyFBA* contains a port *port1~* which is connected to *port1* of *MyCapsule*. Interface variables of *MyFB* which are input variables like *A, B*, and *C* are output variables of *MyFBA*. Interface variables of *MyFB* which are output variables like *D*, *E*, and *F* are input variables of *MyFBA*.
The class symbol of *MyFBA* is given in Figure 7. The declaration of the interface variables has the same syntax like in IEC 61131-3 for function blocks. Ports are displayed like ports of normal capsules. Connections to other capsules or function blocks are only shown in the extended structure diagram.
The second list compartment of Figure 7 shows two operations of *MyFBA*. These operations are investigated in the next section. Keyword *raises* maps corresponding UML-Signals to FB-Signals and vice versa. This mapping is used by the priority-based selection algorithm for conflicting transitions (section 5).

3.2 Behavior

The behavior of FBAs describe how the translation between the function block interface and the capsule interface is done. For this a special language is provided – the FBA-Language.
The FBA-Language defines operations which are called when signals arrive from a port or from the Function Block. We distinguish between operations for the translation from UML-Signals to Function-Block-Signals (FB-Signals) and operations for the translation from FB-Signals to UML-Signals.
In operations of the first category two functions are needed. *Delay(time)* is a function that delays the execution of following commands for the *time* given as a parameter. *WaitFor(bool, time)* is a function that delays the execution of following commands until the Boolean expression given as first parameter evaluates from false to true. The second

parameter is a timeout, which assures that the FBA is not able to hang up. Additionally to these two functions we only need assignments. In assignments access to properties of the FBA class and used data classes is possible. Properties of UML classes are Attributes, Operations, and AssociationEnds. An example operation for the translation of the UML-Signal *sig1* to the FB-Signal *B* is given in Figure 9.

```
ON_UMLSignal (s1: port1.sig1)
BEGIN
  A := s1.attr1;
  B := true;
  WaitFor( F, T#1s);
  B := false;
  A := 0;
  WaitFor( F = false, T#1s);
  A := s1.attr2;
  B := true;
  WaitFor( F, T#1s);
  B := false;
  A := 0;
  WaitFor( F = false, T#1s);
ON_Exception
  B := false;
END_ON_UMLSignal
```

Figure 9. Translation operation for *sig1*

Next we show an operation of the second category for the translation of FB-Signals into UML-Signals. For operations like this additional functions *SendSync(send_signal, receive_signal, timeout)* and *SendAsync(send_signal)* are needed, which send asynchronous or synchronous messages through ports of the FBA. Furthermore declarations of instances of signals are added which are used in calls of the functions *SendSync* and *SendAsync*. *SendAsync* sends an asynchronous message *send_signal*. This asynchronous sending of signal *send_signal* takes no time. If *SendSync* is used instead and *receive_signal* is given as an incoming signal and a timeout is set, the function at

```
ON_FBSignal(E)
SIGNALS
  s2: port1.sig2;
  s3: port1.sig3;
BEGIN
  s2 := D;
  SendSync (s2, s3, T#1s);
  C := true;
  waitFor (E=false, T#1s);
  C := false;
ON_Exception
  C := false;
END_ON_FBSignal
```

Figure 10. Translation operation for FB-Signal *E*

first sends *send_signal* and then waits for *receive_signal*. An example of an operation of the second category is given in Figure 10.
The two operations explained above are typical examples for translation operations of FBAs. All operations consist in their body of the following elements:

- assignments to variables of the associated Function Block
- access to properties of data classes of signals
- calls of the functions
  - Delay(time)
  - WaitFor(bool_expression, timeout)
  - SendAsync(send_signal)
  - SendSync(send_signal, receive_signal, timeout)

The main purpose of the FBA-Language is to give developers of both UML-RT and IEC 61131-3 a common language for the specification of adapters between components of their models. The FBA-Language is not designed to specify behavior of Function Blocks or of capsules. This means that a FBA does not specify what happens after a signal is translated and sent to a capsule or to a Function Block. This is the reason why we left control structures like *IF THEN ELSE* and loops out of the FBA-Language. If an UML-Signal is such complex that the FBA-Language is not sufficient for the translation to FB-Signals, we prefer to redesign the UML-RT interface instead of extending the language. The reason for this is, that the UML-RT system is applied to an existing system. The UML-RT developer should try to keep his design as conform as possible to the design of the existing system.

4. Hardware Solutions

When implementing a FBA the following points have to be considered:

a) How are the Function Block variables synchronized with the FBA variables?
b) How are the translation operations of FBAs invoked?
   The problem here is the invocation of the translation operations of FB-Signal. UML-Signals are triggers for transition of FBAs. To this transitions the necessary operations for the translation of UML-Signals can be added.
c) How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?

Answers to this questions depend very on the hardware connecting the PLC and the IPC. There is no standard way of connecting a PLC and an IPC. Some general examples of doing this are the following:

4.1 Hardware solution 1

If the PLC interface is very simple, then digital inputs and outputs are sufficient. In most cases the IPC must be extended with a digital I/O card. In this solution the FBA is implemented completely at the IPC.

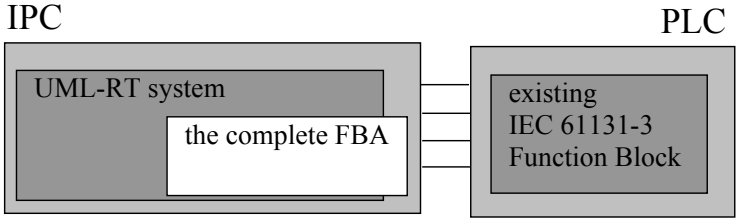


Figure 11. Hardware solution 1: The FBA is only at the IPC

About a) *How are the Function Block variables synchronized with the FBA variables?*
The Function Bock variables can be read and written with the digital I/O card. This can be done by polling or by interrupt techniques.
About b) *How are the translation operations of FBAs invoked?*
Every time a polling function or an interrupt function was invoked, the Boolean expressions of the FB-Signals must be evaluated. If a FB-Signal becomes true, the associated translation operation is invoked.
About c) *How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?*
All functions are implemented and used in the same programming language and environment within the IPC. *Delay* and *WaitFor* could become wait states of a statechart, which are left after a timeout signal of a timer or after the value of a variable has been changed. *SendAsync* and *SendSync* are functions normally provided by the realtime service library of a UML-RT tool.

4.2 Hardware solution 2

A second typical and more important way of connecting a PLC to an IPC is if the PLC uses serial communication over an industrial fieldbus or simply a serial interface like for example RS232 to communicate with an IPC. The IPC uses its existing serial interface or must be extended with a fieldbus interface. The implementation of the FBA then consists of two parts. One part resides at the IPC and the other part at the PLC. Between the two parts a communication protocol must be established within the FBA.

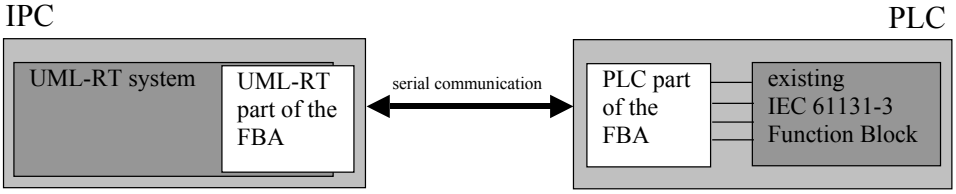


Figure 12. Hardware solution 2: The FBA is distributed over PLC and IPC

About a) *How are the Function Block variables synchronized with the FBA variables?*
For the communication between the PLC and the IPC a special FBA-internal protocol must be developed. Depending on this protocol every changed value of a FB-variable can be sent to the IPC or only meaningful trigger-events. (See also section 5)
About b) *How are the translation operations of FBAs invoked?*
The part of the FBA which is implemented in the PLC is executed in every cycle of the PLC. Each cycle the Boolean expressions of the FB-Signals are evaluated. If a signal becomes true, a message containing all necessary information is sent to the IPC.
Also every cycle the FBA part of the PLC must check if the FBA part of the IPC wishes to send a message.
About c) *How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?*
Delay and *WaitFor* are implemented completely at the PLC part of the FBA in IEC 61131-3 languages. *SendAsync* and *SendSync* are implemented in the IPC part of the FBA. (See also section 6.3)

5. An Execution Model for MyFBA

In this section we explain execution behavior of *MyFBA*. This behavior is a result of a complete FBA-specification given in section 0. It describes when and under which conditions a FBA-operation is processed and in which operational states the FBA may reside. The statechart of Figure 13 shows different abstract states of operation for *MyFBA*. The states are abstract because they must be refined in different ways, depending on the hardware solution (see also section 4). This statechart is an implementation view of the FBA. It doesn't belong to the FBA-Language.

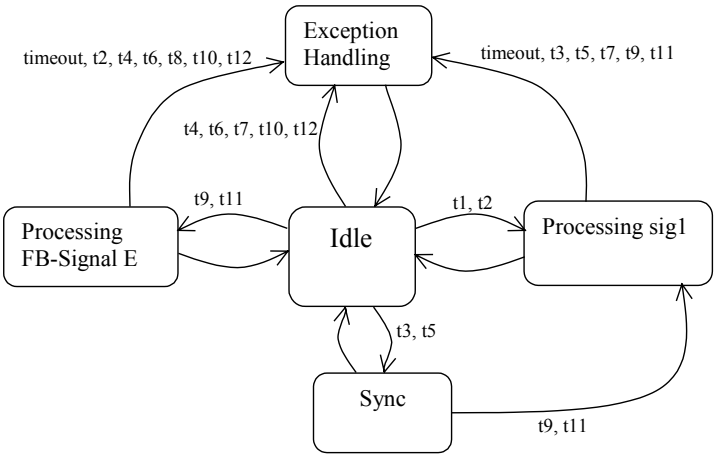


Figure 13. Execution model of MyFBA

The triggers *t1* to *t12* of Figure 13 are defined in Table 1. *MyFBA* has four inputs which may generate events. *Port1* generates the events *sig1* or *sig3* on receiving *sig1* or *sig3*. The input variables *D*, *E* and *F* generate a value-changed event, when the value of a variable has been changed. The events of all inputs are always combined by the logical AND-function. For example, trigger *t1* means that no other event than *sig1* has been occurred. The triggers of transitions in Figure 13 are logical OR-connected.

The central state of Figure 13 is *Idle*. If nothing has to be translated the FBA is in this state. A change-event of variable *D* has no effect in this state. The values of all Boolean variables *E, F, B*, and *C* must be *false*. A change-event of *F* or *sig3* is a protocol exception. This results in a state change to *Exception Handling*. The standard exception handling behavior is to raise an exception message of the operating system. In general, the behavior of this state should be user-defined.

On *t1* or *t2 MyFBA* switches from *Idle* to *Processing sig1*. In this state the FBA-operation

Table 1. Definition of trigger-events

	port1	D	E	F
t1	sig1	-	-	-
t2	sig1	x	-	-
t3	sig1	x	x	-
t4	sig1	x	-	x
t5	sig1	-	x	-
t6	sig1	-	-	x
t7	sig3	-	-	-
t8	-	x	-	-
t9	-	x	x	-
t10	-	x	-	x
t11	-	-	x	-
t12	-	-	-	x

- means no event.
x means value-changed event.
Events on port1 are named like associated signals.

of Figure 9 is executed. A further *sig1* remains in the input queue of *port1* until state *Idle* is reentered. If a deadline is reached or an unexpected event occurred, a transition to state *Exception Handling* fires. In this transition the statements after *ON_Exception* of Figure 9 are executed.
On *t9* or *t11 MyFBA* switches from *Idle* to *Processing FB-Signal E*. In this state the FBA-operation of Figure 10 is executed. A *sig1* remains in the input queue of *port1* until state *Idle* is reentered. With this a kind of extended run-to-completion semantic is reached for the processing states. If a deadline is reached or an unexpected event occurred, a transition to state *Exception Handling* fires. In this transition the statements after *ON_Exception* of Figure 10 are executed.
When in state *Idle* at the same time *sig1* occurs and *E* becomes *true* (*t3* and *t5*), the state *Sync* is entered. Within this transition variable *B* is set to *true*. According to Figure 5 and the priority-mapping given by the FBA (Figure 7) *Sync* is left to *Processing sig1* when *E* is reset.
When implementing hardware solution 2 (section 4.2) every state of Figure 13 must be refined with at least two AND-states, because MyFBA contains two concurrent processes. That's why a FBA-internal synchronization is necessary before one of the trigger-events in Table 1 can be recognized. The next section discusses some technical questions about hardware solution 2 in more detail. Some real time dependent questions, which can be considered at this abstract implementation stage, are discussed in [6].

6. Example for the Fieldbus Profibus-DP and the PLC S7-300

In this section we discuss the hardware solution of section 4.2 realized by a fieldbus of type Profibus-DP and a PLC of type S7-315-DP. The communication between the IPC and the PLC is done over the Profibus-DP. For this the IPC uses a communication processor (CP) called Profibus-CP 5412. The PLC of type CPU315-DP already contains a Profibus-CP.
Like mentioned above, the FBA is implemented in two parts – the capsule part resides at the IPC and the function block part (FB-part) resides at the PLC.

6.1 How are the Function Block variables synchronized with the FBA variables?

Because both the IPC and the PLC are active nodes we need a master-master protocol for communicating over the Profibus. A suitable fieldbus protocol is the FDL (Field Data Link) protocol [8].

```
(1)  void synchronizeAction() {
(2)    ...
(3)    if(SCP_receive(...))
(4)      internalPort.FBSignalE().send();
(5)    ...
(6)  }
```

Figure 14. C++ code fragment

At the IPC the FDL programming interface is provided by a C library with function calls like *SCP_send* and *SCP_receive*. At the PLC the two functions *AG_SEND* and *AG_RECV* are used for FDL-connections. With the FDL-protocol messages can be received either asynchronous or synchronous. The configuration, initialization, and parameter setting of FDL-connections is out of the range of this paper.

```
(1)  FUNCTION_BLOCK MyFBA
(2)  VAR
(3)    my_trig: R_TRIG;
(4)  END_VAR
(5)  ...
(6)  my_trig( E );
(7)  IF my_trig.Q THEN
(8)    ...
(9)    AG_SEND( ... D ... );
(10)   ...
(11) END_IF
(12) ...
(13) END_FUNCTION_BLOCK
```

Figure 15. ST code fragment

As mentioned in section 4.2 the synchronization of the two concurrent processes within MyFBA is done over an internal protocol. This protocol is on top of the FDL-protocol. In our example implementation both sides use polling for recognizing messages of the communication partner. The C++ code fragment in Figure 14 belongs to the capsule-part of the polling mechanism. It calls the function *SCP_receive* to check if the function block part of the FBA wants to send a message with the call of *AG_SEND*. The FB-part of the FBA checks (call of function *AG_RECV*) in every PLC cycle if the capsule part of the FBA wants to send a message with the call of *SCP_send*. The next section explains this aspect again but in more detail.

6.2 How are the translation operations of FBAs invoked?

The translation operations of UML-signals are invoked by the UML-signals itself, after synchronization with the PLC-part was successful.
The FB-signals at first have to be recognized. Then the data of the FB-signal is transferred to the capsule part by a FDL-connection. At the capsule part an internal UML-Signal is generated which triggers the transition which is responsible for the FB-Signal (section 5). This polling is done within the abstract *Idle* state of Figure 13.
The recognition of the FB-Signal is done within the FB-part. The key mechanism is the edge recognition of Boolean expressions. In our example of Figure 10 the Boolean expression consists only of the variable *E*. For edge recognition a function block called *R_TRIG* is provided in [2]. A code fragment of the FB-part of the FBA written in Structured Text [2] is given in Figure 15.
For the explanation of Figure 15 we outline the execution behavior of a PLC in Figure 16. PLC functions are executed in every PLC cycle. At the beginning of a cycle the input

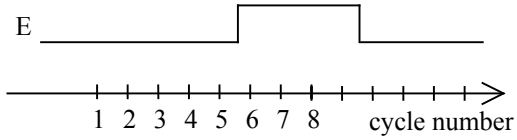


Figure 16. Scanning of input variables of function blocks

variables are read. Line (6) of Figure 15 evaluates in every cycle, if the value of *E* has changed from *false* to *true*. This happens in cycle 6 of Figure 16. Only in this cycle the

output variable *my_trig.Q* is *true*, which is evaluated in line (7) of Figure 15. Then the function *AG_SEND* gives the data of variable *D* to the Profibus system which sends the data over a FDL connection to the Profibus-CP of the IPC.
The time for sending of FDL messages can be greater than the cycle time of a PLC. Furthermore the time interval in which the polling action of the IPC is executed, is in most cases greater than the cycle time of the PLC. This must be considered when FBAs are implemented.
During the design of FBAs we don't need to think about these different cycle times, because a continuous time model is considered. This is a great advantage of FBAs.

6.3 How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?

Delay	For time delays a special function block called *TON* is provided in [2]. (Within the S7-SCL this function block is called *S_ODT*)
WaitFor	The implementation of *WaitFor* is a combination of *R_TRIG* and *TON*. The timer TON is used to generate the timeout.
SendAsync and SendSync	Line (4) of Figure 14 is an example implementation of *SendAsync* with the use of the Rational Rose C++ Realtime Library. For a synchronous message the C++ function *RTOutSignal::invoke()* is provided.

7. Summary and Future Work

Within this paper we have shown that with *Function Block Adapters* the integration of systems designed in UML-RT into an existing PLC environment can be easily specified. The specification of a *Function Block Adapter* is completely plattform-independent. It describes only the "What" should be done for the integration and not the "How". This aspect is very important because the "How" is highly plattform-dependent.
An approach related to our FBA-Language is proposed in the Statemate Approach [3]. In Statemate *reactive mini-specs* are used to specify data-driven activities. Data-driven activities are continuously (cyclic) executed, which is expressed with *TICKs* in a *mini-spec*. Conditions are evaluated in *IF THEN ELSE* statements. In our approach FBA-operations are only executed on associated signal events, which is a different semantic than data-driven activities have. For this reason we introduced the notion of a FB-Signal. Conditions on data-values are evaluated with the *WaitFor* function. The decision if conditions are computed continuously or interrupt-driven is left to the implementation. Whereas data-driven activities are suitable for raw sensor data the FBA-Language is easier to use with IEC 61131-3 Function Blocks. We assume that raw sensor data is computed within a Function Block.
With a specification given in the FBA-Language a developer has an unambiguous description of the requirements for connecting the UML-RT system to the PLC. Because of the simplicity of the FBA-Language both UML-RT developers and IEC 61131-3 developers can understand and validate the specification.
Currently, we are interested in the development of an implementation framework for *Function Block Adapters*. This framework contains

- an integration process,
- class and Function Block libraries,

- design patterns,
- a FBA-Language parser and compiler,
- a simulation environment for validation purposes,
- a modelchecker for verification purposes.

Furthermore we plan to adapt *Function Block Adapters* to IEC 61499. Function Blocks defined in IEC 61499 distinguish between event input and output signals and data input and output signals. These separation would ease our definition of FB-Signals.

8. References

[1] B. Selic and J. Rumbaugh: Using UML for Complex Real-Time System, 1998, http://www.rational.com/products/rosert/whitepapers.jsp

[2] Programmable controllers - Part 3: Programming languages (IEC 61131-3: 1993)

[3] D. Harel and M. Politi: Modeling Reactive Systems with Statecharts, McGraw-Hill, New York 1998

[4] B. Selic, G. Gullekson, P.T. Ward: Real-Time Object-Oriented Modeling. Wiley, New York, 1994

[5] T. Heverhagen, R. Tracht, *Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters*, Proc. IEEE Int. Symp. on Object Orient. Realtime Computing (ISORC2001), May 2-4, 2001, IEEE Computer Society. pages 395-402

[6] T. Heverhagen, R. Tracht, Echtzeitanforderungen bei der Integration von Funktionsbausteinen und UML Capsules, PEARL 2001, Echtzeitkommunikation und Ethernet/Internet (P.Holleczek, B.Vogel-Heuser (Hrsg.)), Informatik aktuell, Springer-Verlag 2001, S. 87-96, in german

[7] T. Heverhagen, R. Tracht, Using Stereotypes of the Unified Modeling Language in Mechatronic Systems, Proc. of the 1. International Conference on Information Technology in Mechatronics, ITM'01, October 1-3, 2001, Istanbul, UNESCO Chair on Mechatronics, Bogazici University, Istanbul, Turkey, pages 333-338

[8] SIEMENS, SIMATIC NET Software NCM S7 for PROFIBUS, Users Guide

Specifying Building Automation Systems with PROBAnD, a Method Based on Prototyping, Reuse, and Object-orientation

Andreas Metzger, Stefan Queins

Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{metzger, queins}@informatik.uni-kl.de

Abstract: In this article, the PROBAnD requirements engineering method, which is specialized towards the domain of building automation systems, is presented. The method bases on object-orientation to handle complexity, reuse to gain efficiency as well as product quality, and prototyping to enable test-based verification and validation early in the development process. To demonstrate the applicability and efficiency of this method, the results of an extensive case study are introduced.

1 Introduction

The development process of embedded real-time systems differs from that of other applications in many ways. One difference can be the small number of identical systems that are created. Especially in the building automation domain a specific system is delivered only once. Another difference lies in the fact that the life cycle of such systems is relatively long. Therefore, the development process needs to be very efficient and traceability has to be established during the whole life cycle, which includes development as well as maintenance. A further property of building automation systems that has to be dealt with is the complexity that results from the integration of control strategies for different physical effects (e.g., illuminance and temperature). This integration is required because of the physical coupling of these effects (e.g., artificial or natural lighting might heat up an area). Additionally, the complexity stems from the huge number of objects that have to be regarded (e.g., the number of components can range from few hundreds to many thousands).

In this article, we focus on the first phase of the development process, the requirements engineering phase. In this phase, the often-vague needs of the stakeholders have to be transformed into formal system requirements. To gain efficiency in performing this step, the development method needs to be specialized towards a specific domain. This specialization allows the definition of the method by describing the different development activities and products in a very fine-grained manner. Further, the specialization permits the precise definition of guidelines for each activity.

The *PROBAnD* method is one example of such a domain-specific requirements engineering method. It relies on the application of three basic techniques: object-orientation, reuse,

and prototyping. Object-orientation is employed to handle the huge number of objects. Reuse allows the reduction of effort by producing products of high quality. Prototyping permits the early validation of the system together with stakeholders as well as the verification of development decisions by developers.

2 Related Work

For the specification of embedded real-time systems, several techniques exist; e.g., Statecharts [Ha90] or real-time UML [Se99], which is a combination of ROOM [Se94] and the universal Unified Modeling Language, UML [Bo99]. Although Use Cases and Interaction Diagrams are widely used in the UML community, they are not qualified for completely and precisely describing the behavior of a system. A notation with well defined semantics is the Specification and Description Language, SDL [Ol97], which therefore is supported by commercial code-generators. In research projects, several of the above techniques have been extended to allow the specification of non-functional aspects. One example is SDL* [Sp97], which adds non-functional annotations to SDL.

Current development methods for embedded real-time systems often base on one of the modeling languages that are depicted above. Some universal methods, like the Unified Software Development Process [Ja99] or the Object-Engineering-Method [Ru01], claim to be applicable in the domain of real-time systems. Such universal modeling methods are specialized for certain development domains by applying derivations of universal modeling notations. Examples for such specialized methods are ROPES [Do01], which is based on real-time UML, and the Real-Time Object-Oriented Modeling Method [Se94], which is based on the ROOM notation.

Despite this specialization, these methods still approach a broad range of problem domains (e.g., the domain of real-time systems), and therefore have to be adapted to specific development domains (e.g., the domain of automotive control systems) before they can reasonably be executed. We have carried out this adaption step, for which we have explored the influence of characteristic properties of the building automation domain. This adaption lead to the definition of the PROBAnD method, which allows the efficient derivation of the requirements of building automation systems.

3 The PROBAnD Method

To introduce the PROBAnD method, an informal description of the underlying process model is given, which includes the most relevant documents and activities (c.f. Fig. 1). This introduction can only give an overview, because the chosen level of abstraction for this article does not expose all development artefacts and fine-grained activities. For a comprehensive and in-depth description of the PROBAnD method please refer to [Qu02].

It should be noted that this process model does not enforce a strict *phase-oriented* execution, where each activity relates to a specific phase and the whole project may only be in one phase at the same time. Rather, a *workflow-oriented* style is preferred, where the activities are assigned to workflows, which can be executed simultaneously.

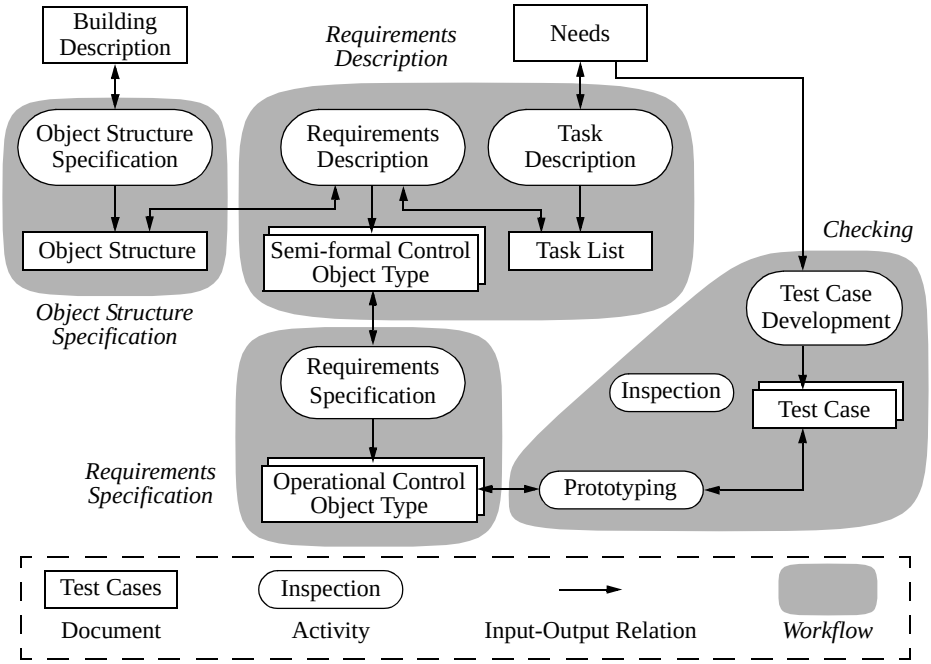


Fig. 1: Overview of the PROBAnD Process

The input for the PROBAnD method is the description of the problem, which can be divided into the *needs* and the *building description*. The needs informally describe the requirements from the point of view of the stakeholders. In the *requirements description* workflow, these needs are split into manageable *tasks*, which depict the requirements from the point of view of the developers and which have to be assignable to one *control object type*.

These control object types form the structure of the system, which is created in the *object structure specification* workflow. An initial *object structure* can easily be derived from the building description because the control object types are most often identical to the building's object types or are a sub-set of these (e. g., lighting needs to be controlled only within the boundaries of one room, and therefore the control object type that is responsible for lighting control can be identified with a room control object type). Only a strict aggregation hierarchy is allowed for the object structure, which follows the hierarchy within the building (e. g., the control object type for a floor aggregates control object types for the rooms of this floor). As requirements engineering proceeds, this control system structure can be refined, as long as the strict aggregation hierarchy of control object types is maintained.

As a guideline for the above activities, we suggest that the tasks that are assigned to a control object type should easily be realizable with the information that is available at that respective level in the hierarchy, which reduces the overall flow of information. Communication between control objects is realized through the exchange of (asynchronous) messages that are only allowed to travel along the aggregation hierarchy. This type of communication together with the strict aggregation hierarchy help maintaining the easy comprehensibility of the specification and allow the creation of documents from a set of few templates to simplify recurring activities.

After the control object types and their tasks have been elicited, the control object types are refined by defining strategies for how the tasks should be solved. These strategies are described in natural language, leading to *semi-formal control object type documents* that are expressed in HTML. Therefrom, each HTML control object type document is transformed into an operational (i.e. formal) document that is specified in SDL [Ol97] in the *requirements specification* workflow. In these documents, the aggregation of control object types is represented by a block hierarchy and strategies are expressed by extended finite state machines, which is a suitable notation for the chosen application domain, as states of control object types can easily be identified (e.g., a luminaire is turned on or off) and the change of states mostly occurs due to events (e.g., the pressing of a button).

Besides inspecting a document for verification purposes, *test cases* are developed that describe typical user interaction scenarios against which prototypes of the system are tested. Further, prototypes are employed for validating the system together with the stakeholders. To benefit from prototyping in the context of an efficient development process, the construction of the prototypes must be possible with little effort. In our approach, this is realized by *generating* prototypes from the set of available documents, which additionally guarantees consistency between the specification and the prototype. Besides SDL documents, HTML documents should also be accessible for prototype generation. To be able to generate prototypes from such semi-formal documents, the PROBAnD method's fine-grained and formally defined product model is employed [MeQ02]. Further, the generated prototypes can be used to produce *traces*, with which the dynamic behavior of the system can automatically be analyzed [Qu99] leading to results that can either provide feedback for requirements engineering activities or that can support decisions during the design phase.

In addition to prototyping, reuse is an important technique to achieve high quality products. Therefore, we allow access to all artefacts that were developed in earlier projects or that are being developed in the current project. These reuse artefacts can range from very small document parts up to complex collections of development products, like the whole system requirements specification. All reuse artefacts are accessible through a so called *dictionary*, which allows the intuitive search for reuse candidates by employing the general structure of a building (e.g., reuse candidates for room control object types can be found by navigating through the dictionary's structure starting at the building and traversing over floor via section to room). Further, domain knowledge that was not packaged in a reusable artefact resides in this dictionary.

4 Case Study

To demonstrate the applicability of the PROBAnD method and to illustrate some of the development documents that were introduced in Section 3, one case study is presented in detail [QZ99]. In this case study an integrated heating and lighting control system for a floor of a university building was to be developed. This floor, which is separated into three sections, consists of 25 rooms of different types, which are connected by a hallway.

The description of the problem consisted of 68 different needs and a building description, which included the description of the installed sensors and actuators. Typical needs were

- "shortly before a persons enters a hallway section, the light should be turned on, if necessary",
- "the use of solar radiation for heating should always be preferred to the usage of the central heating unit", and
- "in the case of malfunctions the system shall provide a stepwise degradation of functionality".

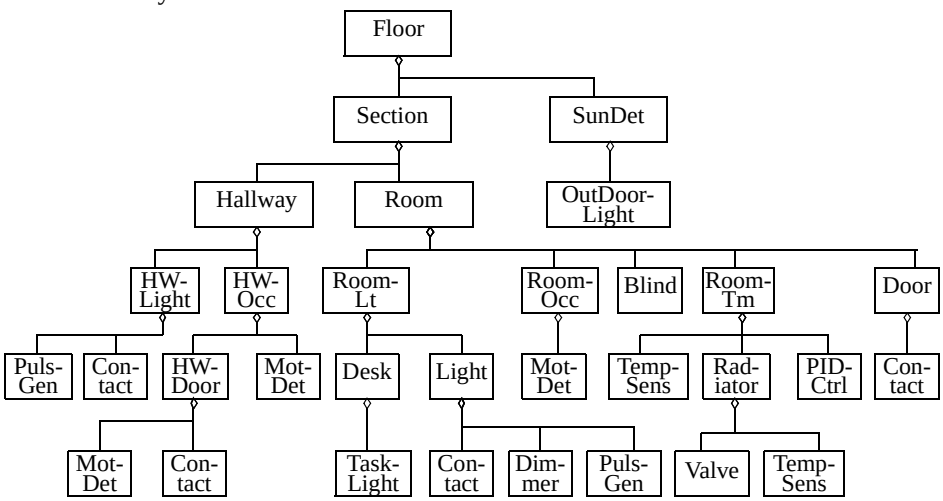


Fig. 2: Object Structure

These needs were split into 126 different tasks, which were assigned to 37 control object types (see simplified object structure in Fig. 2), which are instantiated to 920 instances. To give an impression of the complexity of the control object type documents, the respective HTML files contained 1,280 lines of text. The total effort for the specification of the control system was approximately ten person weeks. Half of the effort was spent for the requirements specification workflow. In this workflow, 118 reuse operations were carried out, where in 81 cases an existing artefact could be reused. Again, to give an idea for the complexity of the resulting SDL specification, the textual representation (SDL-PR) contained 46,000 lines of text. All development products can be accessed on-line at

http://wwwagz.informatik.uni-kl.de/d1-projects/Projects/Floor32/

5 Conclusion

In this article, the efficient requirements engineering method PROBAnD has been presented. The efficiency of this method, which was illustrated by the results of a complex case study, is gained through the application of reuse and generator-based prototyping and the specialization towards a specific domain, the domain of building automation systems.

References

[Bo99] Booch G.; Jacobson, I.; Rumbaugh, J.: The Unified Modeling Language User Guide. Addison Wesley Longman, Reading (Mass.), 1999

[Do01] Douglass, B.P. : Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks and Patterns. 4th Print, Addison-Wesley, Boston (Mass.), 2001

[Ha90] Harel, D.; Lachover, H.; Naamad, A. et al.: STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, Vol.16, No.4, 1990

[Ja99] Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Reading (Mass.), 1999

[MeQ02] Metzger, A.; Queins, S.: Early Prototyping of Reactive Systems Through the Generation of SDL Specifications from Semi-formal Development Documents. In Proceedings of the 3rd SAM (SDL And MSC) Workshop, Aberystwyth, Wales. SDL Forum Society; University of Wales. 2002

[Ol97] Olsen, A.; Færgemand, O.; Møller-Pedersen, B. et al.: System Engineering Using SDL-92. 4th Edition, North Holland, Amsterdam, 1997

[Qu99] Queins, S.; Schürmann, B.; Tetteroo, T.: Bewertung des dynamischen Verhaltens von SDL-Modellen. SFB 501 Report 9/99, University of Kaiserslautern, 1999

[Qu02] Queins, S.: PROBAnD – Eine Requirements-Engineering-Methode zur systematischen, domänenspezifischen Entwicklung reaktiver Systeme. Ph.D. Thesis, Department of Computer Science, University of Kaiserslautern, 2002

[QZ99] Queins S., Zimmermann, G.: A First Iteration of a Reuse-Driven, Domain-Specific System Requirements Analysis Process. SFB 501 Report 13/99, University of Kaiserslautern, 1999

[Ru01] Rupp, C.: Requirements-Engineering und -Management. Carl Hanser-Verlag, München, 2001

[Se99] Selic, B.: Turning clockwise: Using UML in the Real-time Domain. Communications of the ACM, Vol. 42, No. 10, 199; pp. 46–54

[Se94] Selic, B.; Gullekson, G.; Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York, 1994

[Sp97] Spitz, S.; Slomka, F.; Dörfel, M.: SDL* – An Annotated Specification Language for Engineering Multimedia Communication Systems. Sixth Open Workshop on High Speed Networks, Stuttgart, 1997

An Isomorphic Mapping For SpecC In UML

Jorge L. Díaz-Herrera†, Hanmei Chen††, and Rukhsana Alam††.

†B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
20 Lomb Memorial Drive, Rochester, NY 14623
Telf: +1 585-475-4786; FAX: +1 585-475-4775; E-mail: jdiaz@gccis.rit.edu

†† School Computing and Software Engineering
Southern Polytechnic State University
1100 South Marietta Parkway, Marietta, GA 30060-2896

Abstract: An isomorphic UML mapping of SpecC syntax and its semantic preserving transformation is presented. SpecC is one of several competing efforts to deal with the system-level specification and design of embedded systems. We are in the process of providing several mappings, into what is collectively known as YES-UML. This is a broad-spectrum notation built as a series of extensions to UML to support existing modeling concepts. The idea is to use UML as a general modeling language and define isomorphic mappings to existing notations supporting various modeling techniques. YES-UML models are connected via UML implementation technology such as XMI. In this paper we only focus on our isomorphic mapping from SpecC to UML.

Keywords: embedded systems, system-level notations, SpecC, UML extensions.

1 Introduction

Typically, embedded systems are specified using different modeling techniques. As a consequence, many languages and specialized notations are in use today corresponding to differing viewpoints and to the various aspects of embedded systems specification, design, and synthesis. However, no individual notation is, by itself, entirely satisfactory for all aspects of embedded systems development.

There are several notational issues when designing embedded systems, in particular, the level and the kinds of abstraction being specified. On the one hand, the level of abstraction ranges from high-level system concepts all the way down to register-transfer logic (RTL) specifications. On the other hand, we must deal with several kinds of abstractions as systems' components include both hardware and software blocks, and their interconnections. Also, components for both hardware and software blocks are typically specified both, from a structural as well as a behavioral viewpoint. The former refers to physical decomposition, whereas the latter to functional, state-base decomposition.

Recently we are witnessing a strong movement towards a *single system-level design notation addressing both structural and behavioral specifications of both hardware and software components, and their intercommunication mechanisms* [Mo00]. Four

main competing efforts are underway, namely the groups working on SpecC [Dö98], SDL [ITU99a], System-C [SC02a], and Rosetta [SL02a]. Collectively, these notations do provide a rich and powerful set of modeling formalisms and tools. In another front, several researchers are seriously looking at UML [OM00a] as an integrating model, exploiting its extension mechanisms to incorporate relevant system-modeling concepts.

For the Yamacraw Embedded Systems (YES) project [Dm00], we want to be able to reuse existing Hw/Sw blocks. For this, we must be able to capture the design information in a consistent and "standard" form, in such a way that engineers other than the original designers can integrate components. Thus, we need to be able to reuse designs regardless of the notation in which they are specified.

Previous attempts to address this multiple-notation and analyses problem have taken two avenues, a *co-simulation* approach based on a common backplane, and a *compositional approach* based on a single, **"**integrated**"** model. In the former the various analysis tools corresponding to the different notations are cognizant of the semantics of an underlying common backplane. The basis for the second, compositional, approach is the translation of all different notations into a single "integrated" model with its own analysis tools.

Our approach is a variant of the compositional method. We propose a broad-spectrum notation by combining levels of abstraction –vertical integration, and heterogeneous conceptual models –horizontal integration, built as a series of extensions to UML. The idea is to use UML as a general modeling language augmented with syntactic isomorphic mappings and corresponding semantic homomorphism from existing notations supporting various modeling techniques. The mapping works in two ways: at the front-end, analysts deal directly with UML models, whereas at the back-end, designers are able to use their conventional analyses tools that are "seamlessly" connected via UML implementation and information interchange technology such as XMI [OM00b].

We performed a complete isomorphic mapping of SpecC syntax and its semantic preserving transformation into UML via XMI. To investigate its practicality, we built a parser to convert textual SpecC specifications into XMI files, and another parser that takes XMI files and generates SpecC code. The XMI mapping corresponds to the "equivalent" UML SpecC specification, ready for integration with other models, and/or for further processing by analysis tools such as model-checking programs. Indeed, our work complements other YES efforts that use XMI files to interface to model checking tools [BBJ01]. Fig. 1 illustrates the overall concept.

The rest of the paper is organized as follows: Section 2 provides an overview of system-level notations and related work; section 3 describes our UML mapping to

SpecC; section 4 summarizes the tools we built, and section 5 contains the conclusions and further work.

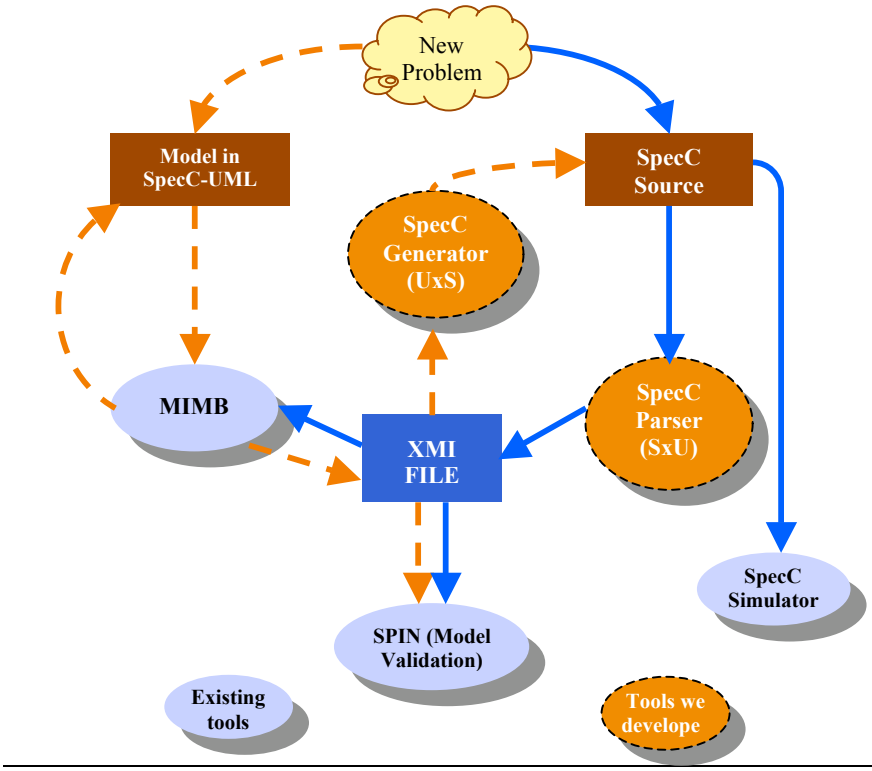


Figure 1: The SpecC-XMI-UML toolset (SxUxS, pronounced "Zeus")

2 Embedded Systems Notations

In what follows we present a summary of each of the most prominent proposed system-level notations. We briefly introduce four system-level notations being subject of intense study lately[1]. The notations are

- **SpecC**, a C-based executable high-level specification and design language;
- **SDL**, a formal system specification and description language;
- **Rosetta**, a "flexible" language devised by the international VHDL's System Level Design Language committee supporting heterogeneous modeling needs; and

[1] For a more complete description see [Di02].

– **SystemC**, which is basically a set of C++ specialized libraries for specifying and designing embedded systems.

The SpecC language evolved around 1997 from the more graphical notation known as SpecCharts [Ga94], by the integration of system-level concepts with the C programming language. SpecC is not an object-oriented language since many of the typical object-oriented features are not present such as inheritance, polymorphism, etc, although the notion of "behavioral types" is used, and it is meant to be more of the concept of abstract data types than classes. SpecC was designed to directly support an embedded development methodology that goes all the way from high-level system specification, allocation and partitioning, to the RTL level and co-simulation. SpecC integrates structural as well as behavioral specifications well.

The Specification and Description Language (SDL) is a much richer, more widely used, and older language than SpecC. In conjunction with the accompanying Message Sequence Chart (MSC) [ITU99b], SDL allows the specification to implementation of discrete, reactive real-time systems. SDL has evolved over the years to accommodate newer concepts and technology. The current standard, SDL-2000, has support for object modeling and implementation. The Z.109 document defines SDL as a UML profile giving the mapping of UML concepts, and an isomorphic mapping has already been done [SR01].

The VHDL's international System Level Design Language committee [SL02b], a worldwide standards initiative, is actively developing an interoperable language environment for the specification and high-level design of microelectronics-based systems, focusing on addressing system-on-a-chip designs. The Rosetta language [AB00] under development[1] provides modeling support for different design domains. Each domain theory provides a semantic and representational framework, including data, computation, and communication models for describing systems *facets*. A facet is a model of a component or system that provides information specific to a domain of interest. Domains provide modeling abstractions for developing facets and components. Constraint modeling, discrete time modeling, continuous time modeling, finite state modeling, and operational modeling are typical modeling domains.

Finally, there is SystemC [Ar00]. SystemC is a C++ modeling platform, made available by leading EDA, IP, semiconductor, systems and embedded software companies, as part of an initiative known as the "Open SystemC Initiative" (OSCI) [SC02b]. SystemC is not a new notation like the other three, but a methodology backed by a collection of C++ class libraries. These libraries support embedded system design work, and help to specify, simulate, and optimize system designs at

[1] Named after the Rosetta stone, which "contained text in three different alphabets and thus helped in understanding each one."

higher levels of abstraction (including performance analysis as well as behavior correctness) [SG00].

2.1 UML notations

Today, several researchers are seriously looking at UML as an "integrating" model, exploiting its extension mechanisms to incorporate relevant system-modeling concepts. It is worth mentioning that UML itself is an attempt to unification of several orthogonal object-oriented modeling elements such as data (classes), behavior (states), and execution flow (actions), each with different notations and semantics. The UML definition, however, as it stands at the time of writing, is not entirely satisfactory for modeling applications in the real-time embedded systems domain. It lacks sufficient semantics in the critical concepts[1].

UML includes built-in *extension mechanisms* that allow refinements of the notation to include concepts found in a particular domain; a coherent set of extensions is known as a *profile*. The **stereotype** is used to expand the semantics of elements already defined in the UML. Stereotypes may have an attached property list of **tag-value** pairs. A tag is the name of a property, and it has a set of given values. Tagged values can be viewed as pseudo-attributes. In addition to tag-values, stereotypes can also have **constraints** associated with them. Compared to tagged values, constraints offer fine-grained specification capabilities written in the *Object Constraint Language (OCL)*, a formal language based on predicate calculus.

Several profiles are being implemented as part of support tools to respond to the real-time systems needs. These are Rose-RT [SR98], ARTiSAN [MC98], Rhapsody [Do99], and ObjectGeode [LE96] among others. These extensions, however, are still fairly low-level, they do not directly address system-level modeling concepts, nor have they full support for all the abstractions of an embedded systems methodology that includes partitioning, Hw/Sw codesign, etc.

Earlier in the Yamacraw project, we experienced the YES-UML idea with a UML mapping of a distributed, event-driven formalism, to show the viability of the approach [Si00]. We mapped ECho [Ei99] to UML concepts and made the corresponding modifications of the Rational Rose-RT toolset to implement the mapping. ECho is an event delivery middleware system whose semantics and structure are similar to those of CORBA's event service specifications. Using the Rational Rose-RT with our definitions, designers can specify models directly using ECho modeling concepts depicted using UML formalisms and notations.

---

[1] At the time of writing, there were four OMG's request-for-proposals to deal with these deficiencies, one dealing with Action Semantics, another with Scheduling, Performance, and Time, and the last two related to quality of service (non-time related), and modeling of complex systems.

3 UML Mapping to SpecC

SpecC integrates system-level concepts and the C programming language. As we said earlier, SpecC is not an object-oriented language. However, it supports the notions of *instantiations*, *interfaces*, and *implementations* without the "complexities" of a full-blown object-oriented language. SpecC specifications are submitted to the various tools for compilation and execution (simulation).

Fundamentally, SpecC specifies a system through the concepts of *behaviors* that interact via *channels* through *ports* and *interfaces*. There is a clear separation between computation and communication where behaviors model computation, and communication is modeled by using shared variables and/or channels. Specifications can be shown both graphically and textually as illustrated in Fig. 2.

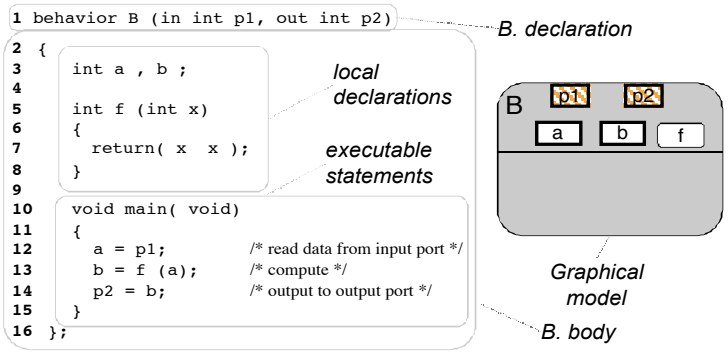


Figure 2: SpecC behavior specification in textual and graphical forms.

3.1 Behaviors, Interfaces, and Channels

A behavior declaration consists of an optional number of ports and/or interfaces. Through its ports, a behavior can communicate directly with other behaviors and/or indirectly with channels. A behavior's body consists of an optional set of instantiations of other behaviors, local variables and local methods, and a mandatory **`main`** method. All methods in a behavior are private, except for the **`main`** method, which is the only public method of a behavior. The **`main`** method is invoked whenever an instantiated behavior is executed. The completion of the **`main`** method determines the completion of the execution of the behavior.

Structurally, behaviors can be broken down into sub-behaviors and these into sub-sub-behaviors, and so on. Designs are specified in a hierarchical manner using basically top-down functional decomposition. A hierarchical network of interconnected behaviors describes the functionality of a system (See Fig. 3). A behavior is called a composite behavior if it contains instantiations of other

behaviors. Otherwise, it is called a leaf behavior; leaf behaviors specify basic algorithms (see section 3.3 below).

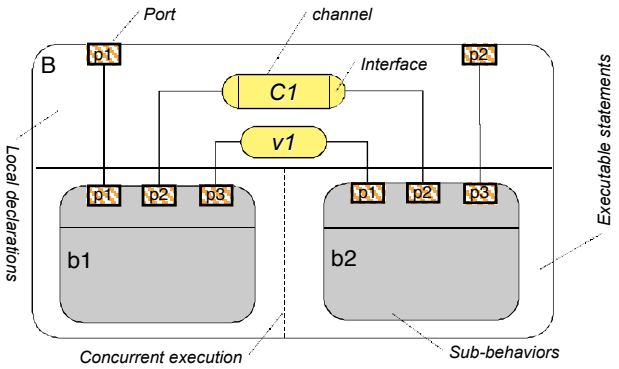


Figure 3: SpecC sub-behaviors, channels, and variables connected via ports.

The most obvious mapping of behavior into UML is a stereotyped classifier **active object <<behavior**>> with the following constraints**:**

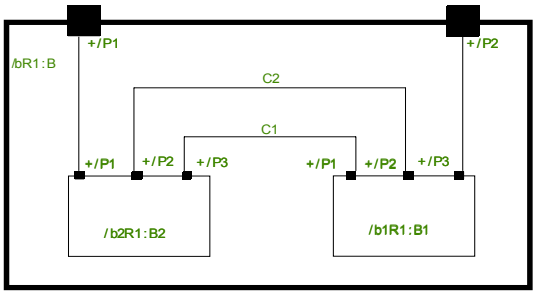
- This classifier must not export any attributes or operations, except for the mandatory **`main`** method;
- There would be a new compartment for the list of communication ports.
- Except for the top behavior, sub-behaviors are actually instances of behavior types.
- A behavior is described from two points of view, namely a logical view and a physical view. For the former we use UML class diagrams as usual. For the latter, we can use UML collaboration diagrams, showing the internal implementation of composite objects as instances and interfaces.
- An alternative way to show the physical structure is by using packages and exported interfaces. If we go this route, we will probably need to use components rather than instances; we are currently studying the benefits of both approaches.

The concept of interfaces is also critical in SpecC. ***Interfaces*** are used to connect behaviors with channels in a way that both, behaviors and channels are easily interchangeable with compatible counterparts ("plug-and-play"). An interface is like an abstract type that consists of a set of method declarations. A behavior with a port of interface type can call the communication methods declared in that interface. A SpecC interface is represented by a standard UML **interface stereotype**.

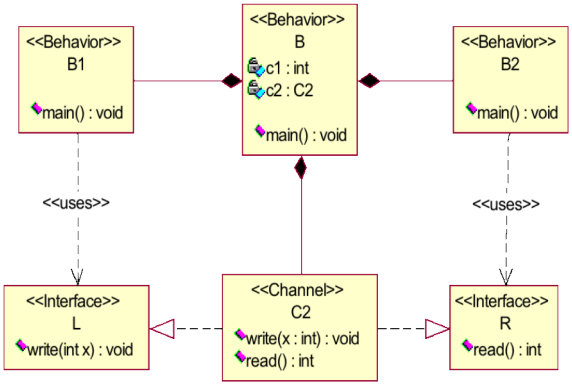
A *channel* is a **passive classifier stereotype** (<<channel>>)**,** which exports only ports and may implement interfaces. The channel (or behavior) that implements the

interface supplies method definitions for these declarations. Thus, for each interface, multiple channels can provide an implementation of the declared communication methods.

Fig. 4 shows an example of the UML mapping corresponding to the various SpecC concepts discussed above. At the top of the figure (a), we present a "special" kind of collaboration diagram, showing the composite object top behavior B, and its corresponding sub-behavior instances (b1 and b2); these correspond to associations by composition shown in the class diagram (b). The class diagram represent the logical view, where behaviors are shown as active object stereotypes; notice the composition relationship to show the inclusion of the sub-behaviors and other passive objects like interfaces and channels (see next section).



(a) Physical view shown as RRT collaboration diagram.



(b) Logical view shown as UML class diagram using SpecC YES-UML stereotypes.

Figure 4: YES-UML SpecC behavior corresponding to example in Fig. 3.

3.2 Algorithmic Specification

A behavior execution is fundamentally independent but not necessarily concurrent. A system specification starts with the execution of the **`main`** method of the **`Main`** behavior. The latter is a composite behavior containing the *test-bench* for a design as well as the *instantiation* of the behavior(s) of the actual design specification. There are three possible execution policies of the statements in a behavior main method. These are sequential, concurrent, and pipelined.

Sequential execution of statements in behaviors is the same as in standard C. A special kind of sequential execution is that of the Finite State Machine (FSM); this allows explicit specification of state transitions. The **`fsm`** construct species a list of state transitions where the *states are actually instantiated behaviors* (see Fig. 5). A state transition is the triple {`current_state, condition, next_state`}, where the `current_state` and the `next_state` take the form of labels and denote *behavior instances*. The `condition` is an expression that has to be evaluated as true for the transition to become valid. The execution of an `fsm` construct starts with the execution of the first behavior that is listed in the transition list (the initial state). Once the behavior has finished, its state transition determines the next behavior to be executed. The conditions of the transitions are evaluated in the order they are specified and as soon as one condition is true the specified next behavior is started. If none of the conditions is true the next behavior defaults to the next behavior listed (similar to a case statement without break). A break statement terminates the execution of the `fsm` construct.

```
{a.main;
 b.main;
 c.main;
}
```

```
fsm {a:{if (x>0) break;
        if(x<=0) gogo b;}
       b:{if (y>0) goto a;
        if(y==0) goto b;}
       c:{ break;}
      }
```

```
Par {a.main;
     b.main;
     c.main;
     }
```

```
pipe {a.main;
      b.main;
      c.main;
     }
```

Figure 5: SpecC execution policies.

Concurrent execution of behaviors can be specified with the **`par`** (parallel) statement. In this case, *every statement in the compound statement block following the* ***`par`*** *keyword forms a new thread of control and is executed in parallel*. (See Fig. 5). The execution of the **`par`** statement completes when each thread of control has finished its execution. **Pipelined** execution, specified by the **`pipe`** statement similarly to the **`par`** construct, is a special form of concurrent execution. All statements in the compound statement block after the pipe keyword form a new thread of control. They are executed in a pipelined fashion (in parallel but obey the specification order).

The **pipe** statement never finishes through normal execution. In the example above (see Fig. 5), the behaviors a, b and c form a pipeline of behaviors. In the first iteration, only "a" is executed. When "a" finishes execution, the second iteration starts, and "a" and "b" execute in parallel. In the third iteration, after "a" and "b" have completed, "c" is executed in parallel with "a" and "b." This last iteration is repeated forever. In order to support buffered communication in pipelines, the piped storage class can be used for variables connecting pipeline stages. A variable with piped storage can be thought of as a variable with two storage places. Write access always writes to the first storage, read access reads from the second storage. The contents of the first storage are shifted to the second storage whenever a new iteration starts in the pipe statement.

Activity diagrams were introduced in UML to "model complex activities taking place within *the system*," they are also very useful to capture algorithmic descriptions at the class level, particularly for showing concurrency and flow within an active object. The activity graph is a special case of a state machine that is used to model processes involving one or more classifiers (is in our case were the actions correspond to the various sub-behaviors). There are other algorithmic constructs in SpecC specifically designed for embedded real-time system specification. We use UML **activity diagrams** to specify the execution that takes place inside a behavior, especially concurrent execution. Fig. 6 below illustrates the mapping to concurrent and piped execution, respectively. The specification of sequential and FSM is similarly done, but not show here for space's sake.

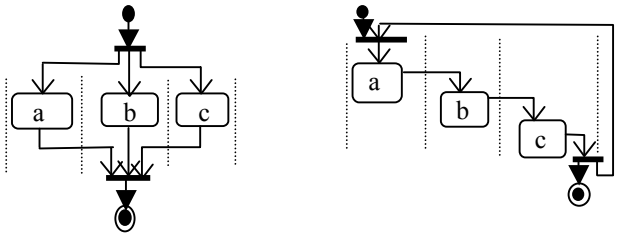


Figure 6: Activity diagrams describing concurrent (left) and pipeline (right) execution

4 Supporting Tools

To automate the mapping process, we created a SpecC compiler that takes SpecC source code as input and generates an XMI file containing corresponding UML model information as output. We employed several existing technologies, including XML Metadata Interchange or XMI [OM00c], JavaCC parser generator [JCC02], and the Meta Integration Model Bridge (MIMB) [MIM02]. In this section, we

briefly introduce these technologies and elaborate on the steps we followed for the implementation.

4.1 UML implementation technology

The **XMI** language was designed to allow the exchange of UML metadata between modeling tools and metadata repositories (based on OMG-MOF) in distributed heterogeneous environments in a standardized way. The eXtensible Markup Language (XML) describes a class of data objects, called XML documents, and, partially, the behavior of computer programs which process them. The overall structure of each XML document is a tree that contains one or more elements, the boundaries of which are either delimited by start/end tag pairs, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. A Document Type Definition or DTD is XML's way of defining the syntax of an XML document. An XML DTD defines the different kinds of elements that can appear in a valid document, and the patterns of element nesting that are allowed. Fig. 7 shows the tree structure of the XMI file, and the nodes under `Foundation.Core.Namespace.OwnedElement`.

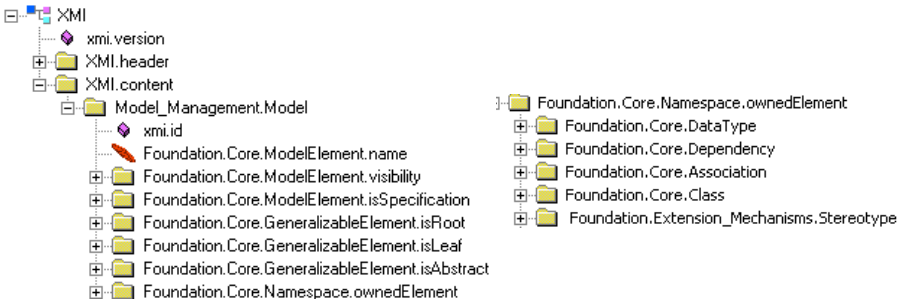


Figure 7: XMI Tree Structure (left) and Sub nodes of (right)

4.2 SpecC Parser Generation

The Java Compiler-Compiler (JavaCC) is a popular parser generator for use with Java applications. It provides several capabilities related to parsing such as tree building, semantic actions, grammar debugging aid, etc. Since SpecC is a superset of C language, we crafted the grammar file by adding the grammatical elements of SpecC to the JavaCC grammar file for C language, already available. *We have donated this SpecC grammar file to the JavaCC grammar repository*. We input the `specC.jjt` file to JavaCC and JavaCC generated a parser of SpecC language as output, which is a Java class file – `SpecCParser.java`. Fig. 8 illustrates the process of obtaining the SpecC grammar file and finally a SpecC compiler.

The JavaCC toolset generates an LL(k) parser. A key characteristic of such parsers is that they decide which production to use by looking at the next 1-k tokens. This means that an LL(k) grammar must not have a point where alternative productions have the same prefix; another restriction is that a production cannot be left-recursive. LL parsers are much easier to understand than LALR parsers, or the more general LR(k) grammar. They are easier to write and debug, and have better error recovery semantics. The entire SpecC LR grammar was published in 1998 [DZG98], and we needed to take out all left recursions and placed additional lookaheads at some decision points.

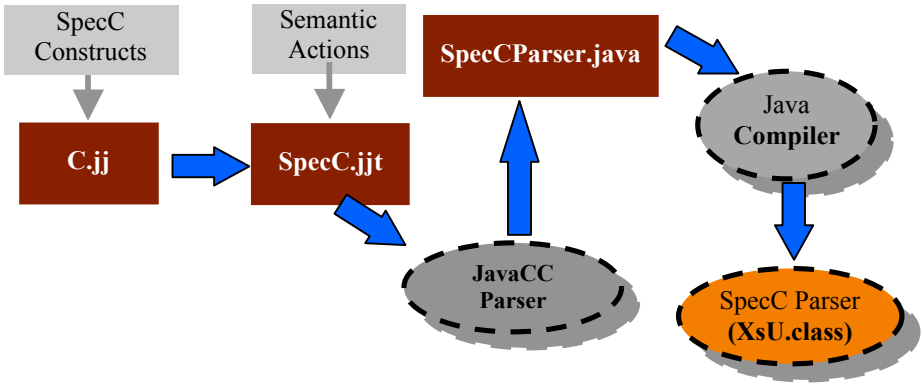


Figure 8: Process of Obtaining a SpecC Compiler

To add the semantic actions to the SpecC parser we developed a set of **Java classes for XMI File Generation**. Instantiations of these classes can be used to generate the nodes of an XMI file. JavaCC allows us to add code for semantic actions to `specC.jj` file and it will automatically copy the code for semantic actions to the SpecCParser.java. Here is an example: when the parser detects a declaration like "`behavior A`" in SpecC source code, it should generate XMI code that contains the information of an active class "`A`", stereotyped as "`behavior`". In the class file SpecCParser.java, each language construct is a function. For example, `external_declaration` is treated in the following way:

```
void ExternalDeclaration() :
{
  InfoOperationNode info = new InfoOperationNode();
}
{
(LOOKAHEAD(FunctionDefinition(info)) FunctionDefinition (info)
  | LOOKAHEAD( Declaration() ) Declaration()
  | LOOKAHEAD( SpecCDefinition() ) SpecCDefinition()
)
}
```

During the process of parsing, when the parser recognizes a SpecC construct that can be mapped to UML, it creates an instance of one of the Java classes and use that object to collect all necessary information from the SpecC source code. Once all needed information has been collected, an XMI node containing the information is generated. The SpecC parser appends that node into the exact position of the XMI tree. As we can see from Fig. 7, we need to generate five types of XMI nodes. In table 1 we listed the XMI nodes need to be generated and the Java class(es) used to generate the corresponding node.

Node Type	Java Class Used to Generate the Node
`Datatype`	`InfoDataTypeNode.java`
`Dependency`	`InfoDependencyNode.java`
`Association`	`InfoAssociationNode.java;` `InfoAssociationEndNode.java`
`Class`	`InfoClass.java; InfoAttribute.java;` `InfoOperationNode.java` `InfoParameterNode.java`
`Stereotype`	`InfoStereotypeNode.java;` `InfoStereotypeRealization.java`

Table 1: XMI nodes and Java classes used to generate the node

In the above code example, `ExternalDeclaration` can be a function definition, a declaration or a SpecC definition. Whenever the parser detects this construct it instantiates an object of class `InfoOperationNode`. If this external declaration is a function definition, the object is passed to the function definition and further down to the leaf level of the syntax tree. As the object flows down through the syntax tree, it collects all information needed to generate the XMI node. Then the node is appended to the XMI node class. This way, semantic actions are combined and the XMI tree is generated.

4.3 XMI File Validation

After we generate a corresponding XMI file for the UML model that contains the same modeling information as the SpecC source code, we pass this file to MIMB, which then generates the corresponding UML model that can be visualized in a suitable UML browser. We performed a manual inspection of the generated output. In order to process XMI data we need an XMI parser. The Xerces DOM Parser was used for this purpose. This Parser builds a hierarchical data structure from the content of the XMI document. It implements the W3C XML and DOM (Level 1 and 2) standards, as well as the SAX (version 2) standard.

We also produced a SpecC code generator from the XMI files. Thus, these two parsers work in both directions, SpecC-to-UML and vice versa, using XMI as an intermediate representation and repository (as shown in Fig. 1). The main functions of the generator (UxS.class) are:

- Take an XMI file as input.
- Parse the XMI file.
- Extract necessary information from the parsed XMI file to generate SpecC code.
- Save the information in a proper data-structure.
- Analyze the information stored according to the XMI specification and UML-SpecC mapping.
- Output the SpecC code.

The XML parser extracts the actual data out of the textual representation and creates either events or new data structures from them. It also checks whether documents conform to the XML standard and have a correct structure. This parsed tree is traversed till all the nodes are explored and necessary information is stored in hashtables and individual data-types. The "key" value of the hashtables is set to the "XMI id" value, which is unique for every element and by which we will be able to find association and relation between channel, behavior and interface. Information related to each SpecC element is stored in the respective data-types. The stored information from the parsed XMI file has to be analyzed for association and relation between SpecC elements. Analyzing of the stored information starts from the Association node. Proceeding further the dependency node has to be analyzed. This results in finding out which Channel implements what interfaces, etc. The Foundation.Core.Class node in the XMI file has all the information that is necessary to output the SpecC code, but to find out the sequence of the elements to be outputted, the Foundation.Core.Class XMI Id's needs to be analyzed. This is done iteratively until all the elements in the Foundation.Core.Class node are explored.

We run several SpecC complete examples through both these tools, and manually inspected the results satisfactorily. We have tested our approach by completely modeling a JPEG encoder [PH01].

5 Conclusions

UML is becoming the dominant industry standard for modeling information systems; it is also being promoted for the real-time embedded systems domain. Several questions remain open with regards to UML support for real-time embedded systems. The approaches proposed so far define two orthogonal models, the class or logical model, and the physical or behavioral model; this separation presents difficulty for model checking. Thus, the combined semantics of the orthogonal class, sequence (and collaboration) and state models it is not defined.

Activity diagrams, those covering control flow and data/object flow, are particularly critical, due to their capability of modeling system-level behavior, yet, it is one of the elements least defined in UML [Bo99]. Detailed semantics of *operations* is also lacking (e.g., non-blocking asynchronous, timed-calls, etc.). There is also some redundancy in the various diagrams and this may lead to inconsistencies.

Efforts are currently underway by the OMG and, independently, by tool vendors, to make UML useful for embedded real-time systems. OMG is working on these and other related issues. According to Selic [Se00], "*the OMG is currently working on a series of standard real-time profiles. An early profile is expected to be available in 2001 and includes modeling time and time-related facilities and services. Two other profiles will be available later, dealing with fault-tolerant systems, and with complex systems architectures.*" It is anticipated that these standard definitions will not be available until sometime in 2002-3.

There seems to be no need to extend the conceptual base of UML since we can use extension mechanisms already in place, specializing appropriate base concepts by adding additional semantics (without violating existing semantics) as stereotypes with constraints. We have demonstrated the approach of extending UML by providing a mapping to the ECho formalism, earlier work, and to SpecC, this work. However, if everyone extends the language in whatever ways they see it fit, we will have a proliferation of extensions. In the mean time, we will continue experimenting with the extension mechanisms, and most specially OCL. A proposal for a "notation integration" based on UML mappings of concepts from existing best-practice notations/techniques is at the heart of our project.

These are synchronization and exception handling mechanisms, and the handling of time and timing behavior. Activity diagrams can also be used to portray these features as usual.

6 References

[AB00] Alexander, P. and D. Barton. "Rosetta provides Support for System-Level design" EETimes.com, special issues, 2000.

[Ar00] Arnout, G. "SystemC standard." Asia and South Pacific Design Automation Conference, 2000. pp 573-577.

[BBJ01] Bobbie, P., Buggineni, V., and Ji, Y., "Model Checking with sf2smv/SMV and Simulation of Parallel Systems in Matlab," Huntsville Simulation Conference (Society of Computer Simulation – SCS), Huntsville, AL, October 3-4, 2001.

[Bo99] Bock, C. "Unified Behavior Models." Journal of Object-Oriented Programming, Vol. 12, No. 5, September 1999.

[Di02] Díaz-Herrera, J. L. "A Survey of System-Level Design Notations for Embedded Systems" In Handbook on Software Engineering & Knowledge Engineering, Ed. Chang. World Scientific Pub. Co., Vol II, June 2002.

[Dö98] Dömer, R., Zhu, J. and D. D. Gajski. The SpecC Language Reference Manual. Technical Report ICS-98-13, ICS, University of California, Irvine.

[Dm00] J. L. Díaz-Herrera and V. Madisetti "The Yamacraw Embedded Software (YES) Methodology", CSIP TR-00-01, ECE Georgia Tech. 31 January 2000. (http://users.ece.gatech.edu:80/~vkm/TR/2000/yesmeth.pdf)

[Do99] Douglass, B. P. Doing Hard-Time: Real-Time Systems with UML. 1999.

[DZG98] Rainer Dömer, Jianwen Zhu, Daniel D. Gajski, "The SpecC Language Reference Manual," UC Irvine, Technical Report ICS-TR-98-13, March 1998

[Ei99] Eisenhauer G. The ECho Event Delivery System. College of Computing, Georgia Institute of Technology. 1999.

[Ga94] Gaski, D.D., F. Vahid, S. Narayan, and J. Gong. Specification and Design.of Embedded Systems. Prentice-Hall, 1994.

[ITU99a] ITU-T Recommendations Z.100, Z.105, Z.107, Z.109 (11/99). www.itu.ch

[ITU99b] ITU-T Recommendations Z.120 (11/99). www.itu.ch or www.itu.int/itudoc/itu-t/approved/z/index.html

[JCC02] JavaCC Grammar Repository http://www.cobase.cs.ucla.edu/pub/javacc/

[LE96] LeBlanc, P. and V. Encontre. ObjectGeode: Method guidelines. Verilog SA, 1996.

[MC98] Moore, A. and N. Cooling. Real-Time Perspective – Overview. (Version 1.0), ARTiSAN Software, 1998.

[MIM02] Meta Integration Model Bridge (MIMB) http://www.metaintegration.net/Products/MIMB/

[Mo00] Moretti, G. "Get a Handle on Design Languages" EDN, June 5, 2000 pp 60-72. (http://www.ednmag.com)

[OM00a] OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000 (http://www.omg.org/)

[OM00b] OMG XML Metadata Interchange (XMI) Specification, Version 1.0, June2000 (http://www.omg.org/)

[OM00c] "OMG XML Metadata Interchange (XMI) Specification", Version 1.1, November 2000

[PH01] H. Padmanabha and J.L. Diaz-Herrera. "System Level Languages SpecC/SystemC and Standards: a Case study with JPEG encoder." SPSU-CS TR April 2001.

[SC02a] SystemC Community: http://www.systemc.org

[SC02b] Gerlach, j. and w. Rosenstiel. "System Level Design Using The SystemC Modeling Platform" http://www.systemc.org

[Se00] Selic, B. "A generic Framework for Modeling Resources with UML." IEEE Computer, June 2000.

[Si00] J Sierra, "UML->ECho Mapping" MS Project report. SPSU, department of Computer Science, Spring 2000.

[SL02a] http://www.inmet.com/SLDL/

[SL02b] Rosetta. VHDL's international System Level Design Language committee. http://www.sldl.org

[SG00] Semeria, L. and Ghosh, A. "Methodology for hardware/software co-verification in C/C++." Asia and South Pacific Design Automation Conference, 2000. pp 405-408.

[SR98] Selic, B. and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. ObjectTime Limited, 1998.

[SR01] Selic, B. and J. Raumbaugh "Mapping SDL to UML" Rational Software 2001.

On The Real Value Of New Paradigms

Theodor Tempelmeier

Department of Computer Science
Laboratory of Real-Time Systems
Rosenheim University of Applied Sciences
Hochschulstr. 1
D-83024 Rosenheim
tempelmeier@fh-rosenheim.de

Abstract. This is a critical assessment of some of the new paradigms of software engineering. The Unified Modeling Language, the notion of design patterns, and some ideas for future and more advanced modelling elements are investigated. This is done from a practical and theoretical point of view, with a focus on real-time and embedded systems development.[1]

1. Introduction

In computer science, new paradigms arise almost continually. In contrast to the real scientific breakthroughs these new paradigms are usually advertised with lots of euphoria, making developers uncritical about possible drawbacks and problems. Even if some new paradigms do not have intrinsic problems, they may suffer from a naive transfer to the domain of real-time and embedded systems development. Instead of such a naive transfer, the specific requirements of real-time and embedded systems development may necessitate a deviation from the original proposals. With this, the Unified Modeling Language, the notion of design patterns, and some ideas for future modelling elements are critically assessed in the following. Of course, this assessment is done with a focus on real-time and embedded systems development only. The author admits that former contact with safety-critical software may have resulted in a bias towards rigour, preciseness, and strong standards in doing real-time and embedded systems development.

It must be pointed out that in the view of the author object-oriented design is definitely the right way of building systems, and this also applies to embedded systems. Encapsulation, abstract data types, etc., and other concepts such as generics or templates, are a must when developing complex embedded systems. This is in fact also the way the more progressive companies do already develop embedded systems. The author is re-

[1] This contribution is an extended compilation of the author's presentations at the OMER and OMER 2 workshops [Te99],[Te01],[Br01]. The author would like to thank the anonymous reviewers of the workshops and the final proceedings for their invaluable and detailed remarks.

luctant, though, to accept inheritance as a dominating design principle. But at least an object-based approach, i.e. encapsulation without inheritance, is the right choice in the author's opinion, without any doubt. Hence, this contribution must not be misunderstood as a criticism of object-orientation in itself.

2. UML for Embedded Systems Development

2.1 Modelling with UML

The Unified Modeling Language (UML) has become the dominating modelling language in software development. UML has its merits, no doubt, in having unified the variety of slightly different but in essence similar notations. For embedded and real-time systems, the benefits of using UML are less clear [Se00]. The author feels that the UML has not been specified with too much concern for embedded and real-time systems development. Pragmatic approaches as in [MM98] underline the specific needs in this area. Generally, thinking about the application of the UML to embedded systems development actually involves questions on different issues:

1. Can something be modelled in UML in principle?
2. Can something be modelled in base UML (i.e. UML without extensions)?
3. Can something be modelled in UML in a "better" way than with previous notations, seen from a pragmatic point of view, i.e. considering only *secondary virtues* such as supply of tools, availability of trained engineers, etc.?
4. Can something be modelled in UML in a better way than with previous notations, seen from a fundamental point of view, i.e. solely with respect to the *concepts* in UML?

The answer to the first three questions is probably a plain "yes". If something can be modelled with certain concepts (say in the framework of ROOM [Se94]) then it can be modelled in UML by defining extensions (stereotypes) which exactly exhibit the same behaviour as the original concepts. UML just serves as an implementation vehicle, in this case. Concerning the second question, one may well assume that anything can be modelled with UML, given the plethora (and vagueness) of concepts in UML. And, of course, availability of tools and the like will profit from the unification process. This contribution only deals with the fourth question, both from a practical and theoretic point of view.

2.2 Software Requirements Specification

In a research and technology project of DaimlerChrysler Aerospace (now EADS) an integrated process for the development of control laws for complex aircraft configurations has been investigated. The flight control software from this project [Ro99b] will be used as an example to evaluate the UML for the requirements specification phase. In a small case study the UML (Version 1.0) has been investigated with respect to speci-

fying requirements for typical flight control software. The results are presented here in the form of three examples:

- definition of control surfaces
- control law block diagrams
- definition of external interfaces

Definition of Control Surfaces.

The main outputs of a flight control system are commands to the control surface actuators. Therefore, a requirements specification for flight control software would typically include a definition of these control surfaces. A possible UML definition for this is given in figure 1. A control surface "is a" primary or a secondary surface (inheritance relation) and so on. And a delta-canard aircraft "has" one rudder, four flaperons, and so on (aggregation or composition relation).

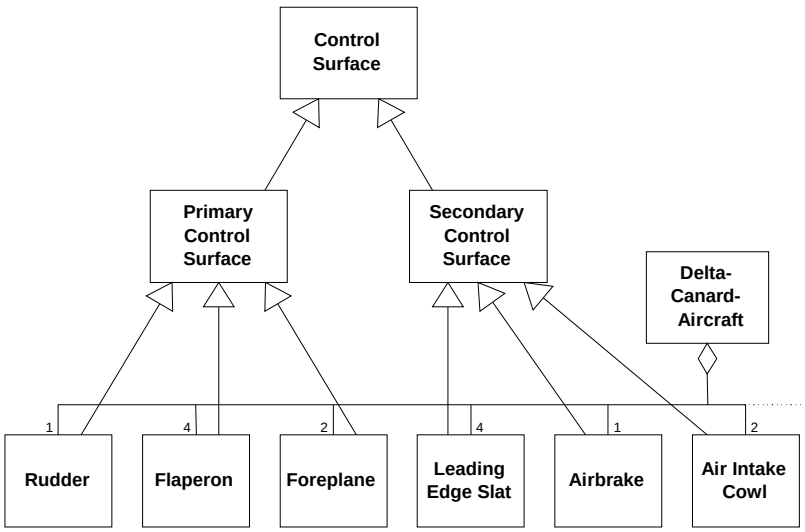


Fig. 1: Control Surfaces of an Aircraft in Delta-Canard-Configuration (UML Diagram)

Figure 2 shows as an alternative a sketch of an aircraft with the primary control surfaces emphasised in white and the secondary ones in dark. It can be seen that a simple figure is sufficient to convey at least the same information as the UML diagram. In fact, a much deeper conception of the physical situation is achieved by figure 2. The author would not consider it very helpful to rephrase such parts of the requirements in UML. (The situation might be different, if the system under investigation would not contain physically visible elements.)

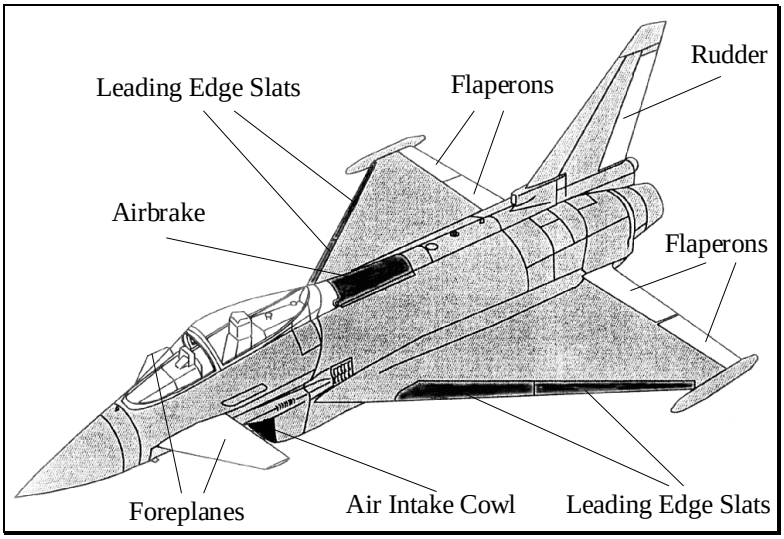


Fig. 2: Control Surfaces of an Aircraft in Delta-Canard-Configuration

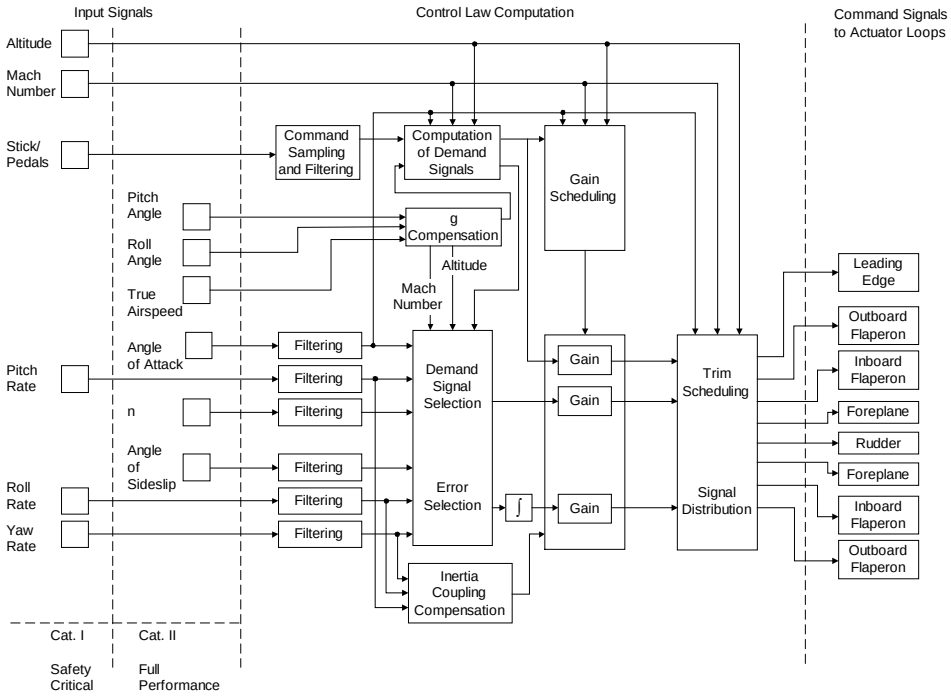


Fig. 3: A block diagram of flight control laws according to [Ka92]

Control Law Block Diagrams.

Requirements for flight control law software are usually defined using control block diagrams, which essentially describe blocks and data flows between them. Each block represents a transformation of input data to output data. The detailed control law computations are described in an algorithmic language, e.g. in FORTRAN. It is essential that this specification is executable, in order to validate the control law design as far as possible before trying to put the real system under test. Obviously, the design of the control laws will go through some iterations, where new ideas and new parameters are tried out and validated (or rejected) until a stable version can be used as requirements for the actual flight control law software. The control block diagram serves as an invaluable aid for maintaining the overview over the control system as a whole. Figure 3 shows an example of such a diagram. If one tries to rephrase such diagrams in UML, the following difficulties arise.

- Obviously, the control block diagram is not a class diagram. Instead, the blocks constitute multiple instances of classes, e.g. the "Filtering" objects. The control block diagram thus rather is an "object diagram". While it is possible to draw object diagrams in UML, only the association relation can meaningfully be used in this case. However, the association relation is only a line drawn between rectangles with almost no semantic information. UML's object diagrams are thus not a suitable alternative for control block diagrams.
- Class diagrams do not help very much in this case, either. They would reveal surprisingly little information, for instance that a notch filter "is a" filter, etc. The complexity of the control block diagram lies in the functionality, i.e. in the contents of the block diagram elements and in their interdependencies. In contrast, UML seems to be more suitable for applications where the complexity lies in the class relationships, like in database applications, for instance.
- One could try to (mis)use other diagrams of UML, e.g. collaboration diagrams, sequence diagrams, or activity diagrams, but the author does not see any advantage in doing so, as compared to using well established control block diagrams.[2]
- Finally, use case diagrams could be used. It is still unclear to the author, whether use cases are just a recurrence of the once condemned functional decomposition models such as Structured Analysis, or whether they also contain some fundamental new ideas.

[2] It is often argued that UML collaboration diagrams could be used instead of control block diagrams or data flow diagrams. However, this is exactly what is meant by the term misuse above. In the UML specification [OM99] it is stated for collaboration diagrams that "... it is important to see only those objects and their interaction involved in accomplishing a purpose or a related set of purposes, ***projected from the larger system*** of which they are part for other purposes. A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes. The identification of participants and their relationships ***does not have global meaning*** [emphases by the author]." In the view of the author this means that collaboration are to be used to describe different scenarios, not the whole system. Furthermore, the "message" arrows in UML collaboration diagrams clearly involve control flow (specifying the sender and receiver of the message). In contrast, data flow diagrams are somehow more abstract, just showing the flow of data and leaving open how this is accomplished (pushing or pulling the data, or even some other mechanism such as using global data). Pragmatically, if the UML is used, one should reinterpret its definition as it is done in [Go00], where "consolidated collaboration diagrams" are used and where "the information passed between objects" is shown to exhibit something similar to data flow diagrams.

Definition of External Interfaces.

Requirements for flight control software must include a definition of the external interface of the software, including the formats of the input and output values. The author considers Ada a good choice for specifying these formats and advocates its use starting with the requirements definition. The following examples repeat some of these probably well-known features of Ada.

```
type switch_values is (neutral, on, off);
   for switch_values use (neutral => 1, on => 2, off => 4);

C_Small_180 : constant  := 180.0 * 2**-11 ;
type T_Fixed_180 is delta C_Small_180 range -180.0..180.0 ;
   for T_Fixed_180'small use C_Small_180 ;
   for T_Fixed_180'size  use 12 ;
```

Such Ada definitions, firstly, have precise semantics according to the language definition. Secondly, the use of the representation clauses ("for ...") allows a precise format definition down to the bit level. For instance, the first definition from above is an enumeration type, the logical values of which are tied to their bit representations in say a digital input or output register. The second example defines a fixed point type. The type covers a range from -180.0 to 180.0 with an increment of the constant `C_Small_180`. This results in a precision of 11 bits plus the sign bit. The for-clauses ensure that the compiler does not do better than requested, e.g. by using smaller increments and more bits than specified with the delta. This example could represent the format of a 12-bit analog to digital converter, for instance. Note the use of a delta which is not a power of two. This represents a common situation in practice. Other possible representation issues, e.g. arrangement of such 12-bit objects within 16-bit words or alignment with respect to memory block boundaries, are not shown here.

Most importantly, using such definitions in the software requirements specification automatically guarantees consistency between (this part of) the requirements with the final code, because a suitable and validated compiler has to implement the representation directives exactly as prescribed by the ISO standard of the Ada language. The author feels this method to be superior to rephrasing such requirements in some formal notation, and then perform proofs of consistency with the final Ada code. On the other hand, using UML would almost certainly be inferior, because there is no semantics and not even a syntax for describing types in UML.

The conclusion from this is as follows. Appropriate models for specification have been used in the engineering domain for quite some time (perhaps in contrast to the domain of business and administrative applications). It can not easily be seen how the UML would provide any fundamental improvement as compared to the current modelling approaches.

2.3 Software Design

The role of a modelling language in the design phase may be manifold. The two most important roles are

- the modelling language is used as design language down to coding, i.e. the modelling language is also used for "programming"
- the modelling language is used for design and the programming language is used for coding

The problem with the first approach is that most modelling languages do not have semantics as precise as it is necessary for programming. The problem with the second approach is that there may be a semantic discrepancy between the modelling language and the programming language (the programming language is in ultimate authority).

From the experience of applying object-based designs to embedded systems and from experience with various modelling languages and tools (e.g. HOOD, [Te98, Te94]), the author takes the following position.

- "Programming" in the modelling language is neither practicable nor reasonable nor desirable.
- The modelling language can be and should be used for a "visualisation" of the design. *This also implies that the modelling language resembles the design concepts within the programming language on a one-to-one, or "isomorphic" basis.* (Naturally, this is only valid if the programming language includes sound design concepts, e.g. programming languages such as assembler are not considered here. Note also that this is not about visualising the implementation, but about visualising (only) those elements in a programming language, which also resemble modelling concepts, e.g. objects, classes, packages, tasks or active objects and the like.)

Obviously, the UML fits the concepts of mainstream languages such as C++ or Java nicely. The author could accept UML for representing designs targeted to these languages, though not all concepts of embedded systems development (outside C++ and Java) can perhaps be described adequately. As an example, quite a few authors resort to symbols for concurrency which are outside the UML [Do98, Hr02, MM98].

If other target languages are used, e.g. Ada for safety-critical systems, the situation is different. Ada has sound design concepts (which are sometimes superior to C++ or Java concepts) which seem to be without counterpart in UML. This gives rise to the weird situation of *design modelling inversion*, where the language for modelling a design is less powerful (in terms of design concepts) than the programming language. The following three examples are given.

- *Protected objects.* Ada's protected objects are not supported by UML. The keyword "protected" has a totally different meaning to the C++ or to Ada95 programmer. What would be the meaning of a keyword "protected" in UML?
- *Hierarchical libraries.* Hierarchical libraries can be realised by nesting, which causes some problems concerning recompilation, etc. Ada95 has a much smarter

scheme for hierarchical libraries which avoids such problems. How are hierarchical libraries handled in UML? Even if semantics in UML were simply tied to the library model of Java, one would still face the question of how to handle the generic hierarchical library units of Ada95, as there are no generics in Java. Further, it would still be open how to distinguish nested hierarchies from "smart" (i.e. only conceptually nested) hierarchies.

- *Abstract data types (in the meaning of a class without inheritance) as distinct from classes (with inheritance).* Ada95 offers both classes without and with inheritance (keyword "tagged"). There are good reasons for this distinction, for instance that one sometimes wants to avoid certain aspects of inheritance in safety-critical systems (see [Is96]), while there is no reason to refrain from using abstract data types. Suggestions to use the Java concept of "final" classes for the purpose of this distinction may be acceptable, but this seems not to be a concept within base UML, version 1.3.

It is possible, of course, to enrich UML with special notes and comments to reflect some additional concepts. Additionally, tools may employ a variety of switches to steer code generation in the desired way. But, this is exactly a repetition of the well-known situation with earlier notations and tools: The designer has in mind an exact idea of his or her design (in terms of the concepts of the target language, say, Ada). He or she then has to understand the design notation (UML), its semantics (maybe defined in terms of another language, say, C++ or Java), and the effects of the code generation switches of the tool (for instance about 50 switches for generating C++ classes in one of the leading UML tools). And then, maybe, the designer will get the design (tailored to his or her target language) he or she had exactly in mind from the very beginning. Such a procedure is hardly acceptable in practice.

As a conclusion, a one-to-one (or "isomorphic") correspondence of design concepts in the modelling language and in the programming language is required (if the programming language itself includes a set of adequate design concepts as it is the case with Ada). This requirement does not seem to be fulfilled by the combination "Ada and UML".

A final warning seems appropriate on the often heard presumption that a design is always to be independent of the programming language and the target operating system (called target environment for short). True, on a coarse level, the central ideas of a design can be independent of the target environment. This is often referred to as logical or abstract design. However, as the design evolves, more precision is added, and the final design will often be dependent of the target environment. Clearly, this does not refer to syntactic details, but to a dependency from the *concepts* in the target environment.

Consider task interaction as an example. The concepts in the target environment may range from simple semaphores to high-level concepts such as Communicating Sequential Processes (CSP). Such different concepts may for instance affect the number of

tasks in a design, especially when some tasks selectively wait for different conditions without knowing which condition is met first. (This works well with CSP, but may be quite difficult, otherwise.) However, in embedded and real-time systems, the number of tasks definitely is a design dimension, because it affects schedulability. The target environment thus does influence the design. To a certain degree, an abstraction layer or "middleware" can be used to encapsulate details of the target environment. However, an all-embracing abstraction layer for all situations has not yet been found. Additionally, a chosen abstraction layer may be unsuitable in a given situation due to performance reasons or it may be inferior to the native concepts in the target environment. So a dependency from different abstraction layers (which depend themselves on different target environments) remains.

As a second example, the class concept is taken. As shown in the following subsection, the semantics of a class in the UML should be different, depending on the target programming language. So how could a design, which is supposed to be independent of the target programming language, be constructed? One could use a UML class diagram, but this would *only look* as if it were language independent. In fact, the same class symbol would employ different semantics, depending on whether C++ or Java code is to be generated. Arguing that a view on such details is small-minded has to be rejected, at least for safety-critical systems where the lives of human beings are at risk.

2.4 Further Comments on the UML

It was after the original publication of the above that the author encountered even stronger criticism of UML. In [Br01] and during the UML «2000» conference [Co00], the lack of a firm semantical basis of UML was pointed out. Some statements of the latter author during his presentation are repeated here to underline how severe the situation is: "... just pictures, no semantics...", "... confused notations ...", "... deeply confused about the meaning of semantics: the semantics section [of the UML definition] is mostly about abstract syntax." It was indicated that a definition of the semantics of UML would have to take into account differing semantics of target languages, i.e. a family of languages were necessary. As an example, different semantics of inheritance in C++ and Java was given in [Co00]. Indeed, even an innocently looking int or integer does have different semantics in C++, Java and Ada, due to a different handling of overflow, as pointed out in [Ro99b]. Hence, the position that the programming language be in ultimate control and that the design modelling language has to represent the semantics of the programming language, is held by the author even more firmly now. But the demand is not only that the semantics of the modelling language must be precisely defined. Rather, from a practical point of view, if the programming language contains sound design elements (as it is the case with Ada), then the modelling language has to represent these elements with the *same* semantics, i.e. in an isomorphic way as requested above.

Concerning the semantics of packages in the UML, criticism has arisen with respect to its vague definition, changing from version to version of the UML specification. This criticism has been confronted by one of the proponents of UML with the blunt announcement, that the semantics of packages in UML would probably change again [Hr01]. Such a situation is completely unacceptable to large-scale and long-term development of embedded systems, especially in a project where the target language offers a stable and near to perfect package concept, as it is the case with Ada. Using UML's fuzzy and inferior package concept in such a situation would result in design modelling inversion, which is clearly to be avoided. To be useful in long-term projects, the UML would have to reach a level of preciseness and stability comparable to the Ada language.

3. Design Patterns

Design patterns have come into widespread acceptance with the book of Gamma et al. [Ga94]. Following this reference, a design pattern may be seen as "a description of communicating objects and classes that are customized to solve a general design problem in a particular context". In the following the concept of design patterns and its application to embedded and real-time systems is discussed.

3.1 The Value of Design Patterns

Design patterns are an extremely valuable concept. There is no need for discussing or questioning the value of design patterns in the view of the author.

3.2 Early Use of "Design Patterns"

The impression that software design patterns have been "invented" in the early nineties is a misconception. Design patterns have been in use especially in embedded and real-time systems long before their widespread publicity, even if they have not been termed patterns explicitly. To give a few early examples from the domain of task interaction, [Bu84] includes "patterns" for dynamic connections between tasks, and [NS88] elaborates on Buffer, Transporter, and Relay Task Intermediary "patterns".

3.3 Domain and/or Target Technology Dependency of Design Patterns

Design patterns are usually domain and/or target technology specific. While some design patterns may be independent of the target technology to be used, many design patterns will be heavily influenced by the actual target programming systems or by the conceptual world of their creators. Thus, in most titles it should read "design patterns for xyz kind of systems" instead of just "design patterns". The following examples illuminate this issue.

- The main reference for design patterns [Ga94] does not even contain the notion of a task or an interrupt service routine (ISR). Obviously, the book is targeted to a domain different from real-time systems, namely to Smalltalk or C++ systems. Hence, one of the most basic design patterns of real-time systems, the Primary/Secondary Reaction pattern (see figure 4), is out of the realm of this book. (Note that this is not at all a case against the book or the authors.)
  It must be emphasised that tasks and interrupt service routines are not just some low-level concepts, but can be implemented in an object-oriented manner. For instance, in Ada protected units are to be used to encapsulate the interrupt service routine [Bu98], and in C++ they can be encapsulated in classes (with some additional implementation effort) as suggested in [Ru98].

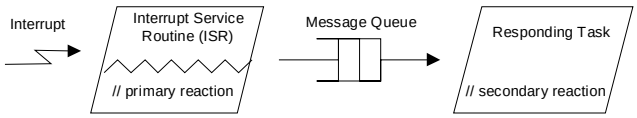


Fig. 4: The "Primary/Secondary Reaction" pattern in real-time systems

- The popular singleton design pattern (which ensures that only one instance of a class exists) may be quite different in module-oriented target languages such as Ada or Modula. A simple data object module (in contrast to a data type module which would represent a class) will automatically ensure the single instance property.
- Templates or generics are not considered with the patterns in [Ga94], because they "aren't needed at all in a language like Smalltalk...". Probably, some patterns, esp. the Template Method pattern, might look different (or might not be needed at all) for target languages with built-in template facilities.
- A Monitor Object pattern [Sc01] may reduce to a simple application of some feature of the target programming language, if the language directly supports such a (or a similar) concept.

3.4 Usefulness of Pattern Structure Diagrams in UML

The UML is widely (almost exclusively) used for visualising designs. It is still unclear, how UML can or should support the design of embedded and real-time systems in general (see [Se00, Do98]). As for the pattern structure diagrams in the (UML), these may be almost completely useless for describing patterns.

As a first example, it is observed that the very basic Primary/Secondary Reaction pattern cannot adequately be expressed in core UML. (Of course, it may be claimed that via UML's extension mechanisms it can be expressed, as almost any concept or symbol is expressible with arbitrary UML extensions.)

Secondly, the usefulness of UML's pattern structure diagrams is demonstrated with the Priority Ceiling Pattern [Do00] (see figure 5). As easily seen, such a pattern structure diagram conveys almost no information at all with respect to the ideas behind the priority ceiling protocol. Such a design pattern diagram is completely useless. This is clearly not the fault of [Do00], but a shortcoming of the UML. A reasonable description of this pattern can of course be given with model elements outside the UML and with plain text [Do00].

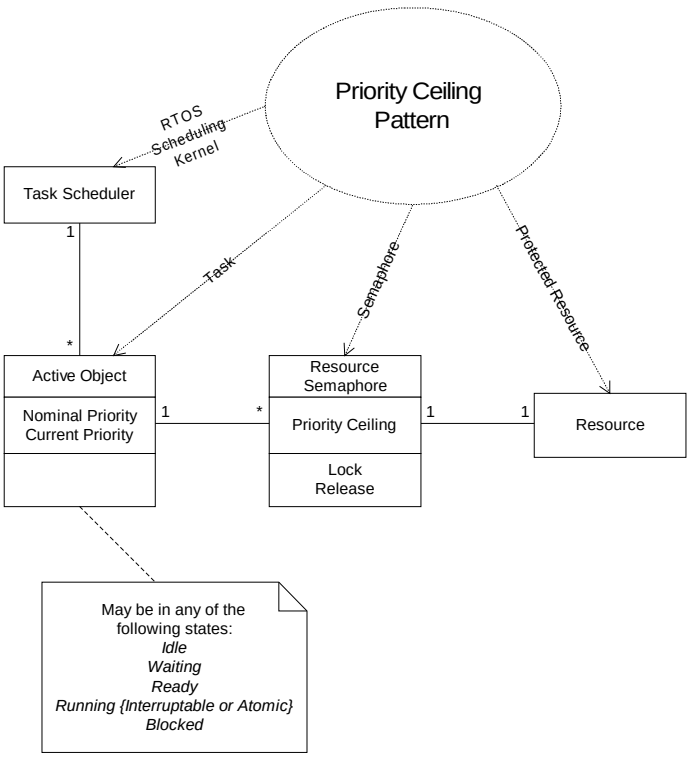


Fig. 5: Priority Ceiling Pattern (taken from [Do00])

3.5 Specific Properties of Embedded Systems

Embedded systems differ from other computer applications in a number of ways. For instance, embedded systems tend to be much more static than, say, a workstation application written in C++ or Smalltalk. This may mean that objects are allocated statically instead of being dynamically created and destroyed, that all program code is burnt in a read-only memory, that heap usage (if any) is minimised, etc. This also has its consequences concerning patterns. Patterns which induce significant run-time overhead may be unacceptable, other patterns may just be unneeded in such a context.

As an example from industrial practice, a seemingly trivial buffer object is considered (figure 6).

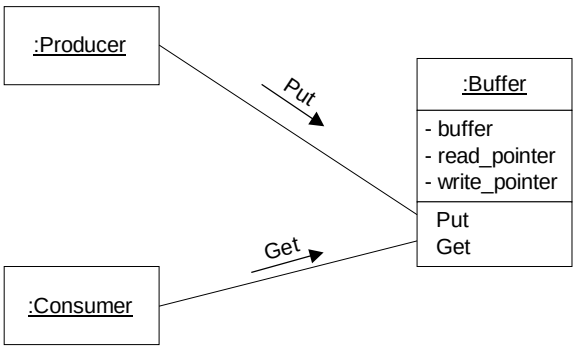


Fig. 6: A simple Producer/Consumer Buffer pattern

However, some typical embedded systems requirements make this an interesting object of study and, eventually, a design pattern:

- One part of this buffer pattern (the data acquisition part) is to be implemented as firmware on a specific hardware board.
- The buffer memory and the read and write pointers have to be at fixed memory addresses, forming the hardware interface.
- The other part of the pattern, i.e. further processing of the data, is implemented as software on a standard CPU.

This may result in a pattern as shown in figure 7. The buffer object is drawn on the edge of the hardware board to indicate that it is partly implemented as firmware, i.e. software on the board, partly as software on the main CPU. The firmware on the board comprises the "put" part, the buffer space, and the read and write pointers of the object, whilst the software on the main CPU implements the "get" part, accessing the buffer space and the read and write pointers via memory-mapped I/O.

Note that firmware and software are usually developed by different teams. Hence, it may be hard to detect such an object or class, spanning the responsibilities of different teams. Even more complexity arises when the firmware team also uses field-programmable gate arrays (FPGAs) to implement parts of the functionality in programmable hardware (e.g. [ST98]), now a common practice.

Further design decisions on the buffer object or class have to be made with respect to synchronisation, concerning genericity, and whether multiple instances should be allowed or not.

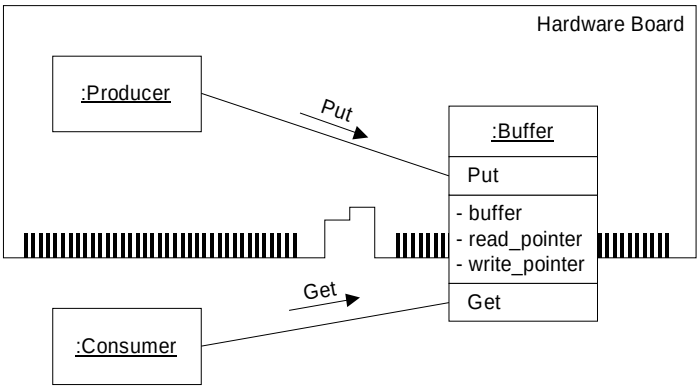


Fig. 7: A Producer/Consumer Buffer pattern with additional constraints of a real embedded system

To extend the above example, let us assume that in fact three consumers are given. These consumers interpret and eventually store or display the same data differently. It should be possible to individually switch these consumers on and off as desired with respect to user input. Note that the number of consumers is fixed at compile time. It was suggested to the author to solve this design task with the well-known observer pattern [Ga94]. This pattern involves an abstract and concrete server, an abstract observer, and concrete observers – the consumers. The consumers are notified by new data, and then updated. In fact, this pattern works for the given situation. However, such a solution is somewhat like an overkill. It involves a dynamically linked list (for the observers/consumers), dynamic memory allocation and deallocation (when switching consumers on or off), and, of course, inheritance, polymorphism, abstract classes, and the like. Apart from the involved overhead such a solution might also look difficult to the inexperienced, because – while being very flexible – the solution is not very explicit (cf. [Fo01] for the benefits of explicit designs). And part of the flexibility, the ability to bring in or take out new observers dynamically, is just not needed here. The author would instead suggest a simplified observer pattern for the given requirements, which would avoid the described drawbacks (see figure 8).

To summarise, design patterns are in many cases target or target technology specific. This means that for embedded and real-time systems the typical solution space with tasks and interrupt service routines, with fixed memory locations, with typical programming languages and operating systems, etc. has to be considered. More design patterns specific to the everyday design problems in embedded and real-time systems are needed!

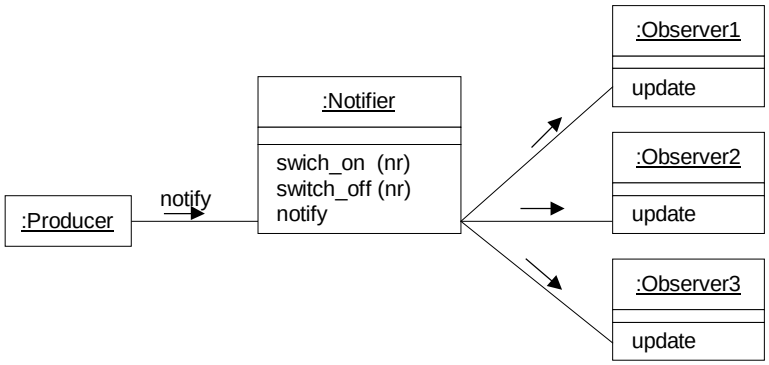


Fig. 8: A simplified observer pattern tailored to the real needs of a specific embedded system

4. The Hot Spots: Future Modelling Elements and Current Practice

During the OMER workshops a variety of ideas about (and requests for) future modelling elements was discussed [Br01, Hr01, and contributions from the audience]. In the view of the author, some of these topics are really the "hot spots" of modelling embedded and real-time systems, recurring again and again. Hence they deserve some special attention.

In the following, these topics will be related to current practice. This will be done along the flight control software project mentioned in section 2.2. As part of this project control laws for a typical fighter aircraft have been implemented in Ada. The resulting software is called COLAda (Control Law Software in Ada) [Ro99a, Ro99b]. Flight control software represents one of the most demanding embedded real-time applications and should be of general interest with the advance of this technology into the automotive industry in the form of x-by-wire applications. So this is an example with a very realistic background.

4.1 Inheritance and Polymorhism

In the design of COLAda object-oriented techniques have been used, where appropriate. Certain control law elements, e.g. filters, have been identified as candidates for objects, and corresponding abstract data types. (e.g. filter types) have been defined. Several objects of these types may be created as required by the control law application –the objects are just instances of abstract data types. The external behaviour of these objects is defined by the operations associated with the abstract data type. Of course all internal data are encapsulated in the objects.

According to a certain terminology, the design might be termed to be object-based, as no type extension ("inheritance") is used. This is for the following reasons:

- Full use of inheritance, in particular polymorphism and dynamic binding (i.e. the use of class-wide types in Ada terminology), may cause certain problems in safety-critical systems (cf. [Is98]).
- There is hardly a need for using inheritance in the context of this project. The cases where variants of certain object types occur (e.g. first order and second order filters) can easily be covered without inheritance.

Though certain aspects of flight control software can be designed and implemented in an object-oriented way, and though the author very strongly favours such an approach, there must be a warning against hypocrisy with respect to object-orientation: The approach "everything is an object" does not seem very helpful. Over-simplistic approaches like in [Co95], where a case study of an object-oriented auto-pilot system is reported on, seem to be impractical for the implementation of real flight control software. And, of course, normal arithmetic operations and static typing are to be used in control law software. This is in contrast to the philosophy of radical object-oriented languages such as SmallTalk and Lisp, for instance, which are considered unsuitable to real-time safety-critical systems.

As discussed during the OMER-2 workshop [Br01], inheritance and its ramifications may indeed not be the desired way of building embedded real-time systems. In the view of the author, composition (of abstract data types) is to be preferred over inheritance in many cases. However, this has been the general guideline in large parts of the Ada and embedded systems community, anyway (e.g. [Ro92]).

4.2 Components and Composition

It has become clear that the class concept is too fine grained for building large scale systems [Br01]. Some larger building block, with a possibility to compose such blocks, is clearly needed. The term component will be adopted here for such building blocks, though this term is not precisely defined. It is unclear to the author, why something new should be necessary here. Rather, the demand during the OMER-2 workshop that one single concept should cover classes and components, seems very meaningful. It is possibly the influence of languages such as C++, which fosters requests for some additional silver bullet component concept. On the other hand, from the author's experience with Ada, there seems to be an at least acceptable concept for structuring large systems. The Ada notion of a package does unify the class concept with the concept of larger building blocks. And the concept does have defined semantics (cf. section 2.4), and packages can be composed, since the advent of Ada95 even hierarchically in a very smart way.

4.3 Concurrency

The failure to adequately address concurrency issues in object-oriented development (e.g. in C++ or in UML) has also been addressed in [Br01]. As for the Ada projects the author has been involved in, there is in fact a concurrency model available. Notwith-

standing possible shortcomings, Ada's concurrency model is somehow very close to Hoare's Communicating Sequential Processes (CSP) – in the Ada95 version even highly performant – giving an level of abstraction and composability the author feels is sufficient. (This holds for concurrency within one processor, not for concurrency in a network of processors.) Note that Java, though more recent than Ada, has an inferior concurrency model, which does not scale – quite in contrast to CSP. Looking at the UML, the issue of concurrency once again gives rise to modelling inversion. That is, appropriate programming languages contain design features which are by far superior to the concurrency features of the "modelling language".

5. Conclusion

As an overall conclusion, all the new and extensively marketed paradigms should be carefully evaluated with respect to exactness and completeness of the definition and concerning the suitability for real-time and embedded systems. In addition, proven and currently available technology must be included in a general assessment.

References

[Br01] Broy, M., Tempelmeier, T., Wirsing, M., Ziegler, J.: *OO Development of Distributed Embedded Systems – A Critical Assessment.* Panel discussion. OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") May 9-12, 2001, Herrsching, Germany. The first author's position statement with similar content in German: Broy, M., Siedersleben, J.: Objektorientierte Programmierung und Softwareentwicklung – Eine kritische Einschätzung. (Object-oriented Programming and Software Development – A Critical Assessment.) Informatik-Spektrum, 25, 1, Februar 2002, p. 3-11.

[Bu84] Buhr, R.J.A.: *System Design with Ada.* Prentice-Hall, Englewood Cliffs, 1984.

[Bu98] Burns, A., Wellings, A.: *Concurrency in Ada.* 2nd Ed. University Press, Cambridge 1998.

[Co95] Coad, P., North, D., Mayfield, M.: Object Models. Strategies, Patterns, and Applications. Yourdon Press, Englewood Cliffs, 1995.

[Co00] Cook, S.: *The UML Family: Profiles, Prefaces and Packages.* In: Evans, A., Kent, S., Selic, B.: «UML» 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings of the Third International Conference. York, UK, October 2000. Lecture Notes in Computer Science 1939. Springer, Berlin 2000. p. 255-264.

[Do98] Douglass, B.P.: *Real-Time UML. Developing Efficient Objects for Embedded Systems.* Addison-Wesley, Reading 1998.

[Do00] Douglass, B.P.: *Real-Time Design Patterns.* White Paper, I-Logix. On the internet http://www.ilogix.com. July 2000.

[Fo01] Fowler, M.: *To be explicit.* IEEE Software, November/December 2001, 18, 6, 10-15.

[Ga94] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading 1994.

[Go00] Gomaa, H.: *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley, Reading 2000.

[Hr01] Hruschka, P., Björkander, M., Färber, G., Kopetzky, V.: *The Missing Concepts of UML for Designing Embedded RT Systems.* Talk Show. OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") May 9-12, 2001, Herrsching, Germany.

[Hr02] Hruschka, P., Rupp, C.: Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML. (Agile Software Development for Embedded Real-Time Systems with the UML. In German). Hanser, München, 2002.

[Is98] ISO/IEC: *Working Draft 3.8 - Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems.* ISO/IEC PDTR 15942, ISO/IEC JTC 1/SC22/WG9, October-29, 1998.

[Ka92] Kaul, H.J.: *Flugsteuerungssystem Jäger 90* (Flight Control System of the Fighter 90. In German). In: G. Bürgener (Schriftleitung). Jahrbuch 1992 III der Deutschen Gesellschaft für Luft- und Raumfahrttechnik e.V. (DGLR). Deutscher Luft- und Raumfahrtkongreß 1992. DGLR Jahrestagung, Bremen, 29. September – 02. Oktober 1992. Deutsche Gesellschaft für Luft- und Raumfahrt e.V. (DGLR), Bonn 1992.

[MM98] McLaughlin, M.J., Moore, A.: *Real-Time Extensions to UML. Timing, concurrency, and hardware interfaces.* Dr. Dobb's Journal, December 1998.

[NS88] Nielsen, K., Shumate, K.: *Designing Large Real-Time Systems with Ada.* Intertext Publications & McGraw-Hill, New York 1988.

[Om99] OMG: *Unified Modeling Language Specification*. Version 1.3, June 1999. On the internet or on CDROM in: Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addisson-Wesley, Reading 1999.

[Ro92] Rosen, J. P.: *What Orientation Should Ada Objects Take?* Communications of the ACM, 35, 11, November 1992, p. 71-76.

[Ro99a] Rosskopf, A.: *Development of Flight Control Software in Ada - Architecture and Design Issues and Approaches.* Ada-Europe'99. International Conference on Reliable Software Technologies. June 7-11, 1999. Santander, Spain. Springer Lecture Notes in Computer Science, 1622. pp. 437-449. Springer, Berlin 1999.

[Ro99b] Roßkopf, A., Tempelmeier, T.: *Aspects of Flight Control Software - A Software Engineering Point of View.* 24nd IFAC/IFIP Workshop on Real-Time Programming, Schloss Dagstuhl, Saarland, Germany, May 31 - June 2, 1999. Pergamon, Elsevier Science, Oxford 1999. Also in: Control Engineering Practice, June 2000, 8 (2000), p. 675-680.

[Ru98] Rusch, D.G.: *Encapsulating ISRs in C++*. Embedded Systems Programming, February 1998.

[Sc01] Schmidt, D.C.: *Monitor Object – an Object Behavior Pattern for Concurrent Programming,* (updated October 10th) C++ Report, SIGS, planned to appear 2000. On the internet: http://www.cs.wustl.edu/~schmidt/patterns-ace.html. February 2001.

[Se94] Selic, B., Gullekson, G., Ward, P.T.: *Real-Time Object-Oriented Modelling.* Wiley & Sons, New York 1994.

[Se00] Selic, B., Burns, A., Moore, A., Tempelmeier, T., Terrier, F.: *Heaven or Hell? A "Real-Time" UML?* In: Evans, A., Kent, S., Selic, B.: «UML» 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings of the Third International Conference. York, UK, October 2000. Lecture Notes in Computer Science 1939. Springer, Berlin 2000. pp 93-100.

[ST98] Schrott, G., Tempelmeier, T.: *Putting Hardware-Software Codesign into Practice.* Control Engineering Practice, 6 (1998) 397-402. Volume 6, Issue 3, March 1998

[Te94] Tempelmeier, T.: *An Overview of the HOOD Software Design Method.* In: Real Time Computing. NATO ASI Series F, Vol. 127. Halang, W.A. and Stoyenko, A.D. (eds.). Proceedings of the NATO Advanced Study Institute on Real Time Systems, Sint Maarten, Dutch Antilles, October 5-17, 1992. Pages 726-734. Springer, Berlin 1994.

[Te98] Tempelmeier, T.: *Hierarchical Object-Oriented Design (HOOD) – Die Software-Entwurfsmethode der europäischen Raumfahrtbehörde ESA.* (Hierarchical Object-Oriented Design (HOOD) – The Software Design Method of the European Space Agency ESA. In German.) Kolloquiumsvortrag an der Universität Oldenburg, 29.6.98. On the internet: http://www.fh-rosenheim.de/tempelmeier.

[Te99] Tempelmeier, T.: *UML is great for Embedded Systems – Isn't it?* In: P. Hofmann, A. Schürr (eds.): OMER ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings. May 28-29, 1999, Herrsching (Ammersee), Germany. Bericht Nr. 1999-01, Mai 1999. Universität der Bundeswehr München, Fakultät für Informatik.

[Te01] Tempelmeier, T.: *Comments on Design Patterns for Embedded and Real-Time Systems.* In: A. Schürr (ed.): OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings. May 9-12, 2001, Herrsching, Germany. Bericht Nr. 2001-03, Mai 2001. Universität der Bundeswehr München, Fakultät für Informatik.

State Machine Modeling: From Synch States to Synchronized State Machines

Dominikus Herzberg *
Ericsson Eurolab Deutschland GmbH
Ericsson Allee 1
52134 Herzogenrath, Germany
Dominikus.Herzberg@eed.ericsson.se

André Marburger
Aachen University of Technology
Department of Computer Science III
52074 Aachen, Germany
marand@cs.rwth-aachen.de

Abstract: To synchronize concurrent regions of a state machine, the Unified Modeling Language (UML) provides the concept of so-called "synch states". Synch states insure that one region leaves a particular state or states before another region can enter a particular state or states. For some application areas, it is beneficial to synchronize not only regions but also state machines. For example, in data and telecommunications, a pure black box specification of communication interfaces via statechart diagrams gives no adequate means to describe their coordination and synchronization. To circumvent the limitations of the UML, this paper presents the concepts of Trigger Detection Points (TDP) and Trigger Initiation Points (TIP); it allows a modeler to couple state machines. The approach is generic, easy to extend and smoothly fits to the event model of the UML; it could also substitute the more specific concept of synch states.

1 Introduction

The problem of synchronizing concurrent state machines raised as an issue in a research project at Ericsson [9]. Concerned with architectural modeling of telecommunication systems, we developed a ROOM (Real-Time Object-Oriented Modeling) [20] like notation (see [8]) but were soon confronted with the question of coupling interfaces: How do we model the interaction between interfaces (or ports) of a single component without referring to its internals? The intention was to describe an architecture in a black box manner, though being capable to understand and simulate interface coordination and synchronization. In

*This work is being funded by Ericsson and is run in cooperation with the Department of Computer Science III, Aachen University of Technology, Germany.

other words, the question was how to properly couple the individual state machines, which specify the interface behavior of a single component.

Independent of this investigation, an Ericsson internal study on the use of modeling languages for service and protocol specifications exactly points out the same problem. It shows that the coupling problem is of theoretical as well as practical relevance. It is also one of the reasons, why modeling languages like the UML (Unified Modeling Language) [17] have not successfully penetrated the systems engineering domain, yet. System designers of data and telecommunication systems do not find reasonable support in today's modeling languages for their problem domain [7].

In the following two subsections, the telecommunication background is introduced and the problem is described in more detail. Subsequent sections discuss the proposed solution: In section 2 we elaborate on the model presented in subsection 1.1 in form of a case study. There, we study the TCP (Transmission Control Protocol) layer of a data communication system and show how the external interfaces can be described by a Finite State Machine (FSM) each. Section 3 discusses the coupling problem from different perspectives and demonstrates how FSMs can be synchronized via Trigger Detection Points (TDP) and Trigger Initiation Points (TIP). The implementation of a prototype, verifying the TDP/TIP concept, indicates how the UML could incorporate the TIPs and TDPs as extensions and supersede synch states; this is subject to section 4. Finally, section 5 closes with some observations and conclusions.

1.1 Background

On an architectural level, any data or telecommunication system can be structured according to two different directions of communication, "vertically" and "horizontally". "Vertical" communication refers to the exchange of information between layers. The "point" at which a layer publishes its services for access to an "upper" layer is called *Service Access Point* (SAP). "Horizontal" communication, on the opposite, refers to the exchange of information between remote peers. Remote peers are physically distributed, they reside in different nodes, and communicate with each other according to a protocol. We call the "point" describing the protocol interface *Connection Endpoint* (CEP). Note that the concept of a protocol is well-known and generally defines a set of messages and rules (see e.g. [2, p.191]); however, it has a special meaning in data and telecommunications. Whereas software engineers associate a reliable, indestructible communication relation with the term "protocol", data and telecommunication engineers are faced with the "real" world: They have to add error correction, connection control, flow control and so on as an integral part to the protocol. A communication relation between remote peers can always break, be subject to noise, congestion etc. This is the reason why communication engineers introduced protocol stacks, with each protocol level comprising a dedicated set of functionality, thereby "stackwise" abstracting the communication service. These stacks naturally give means to "vertically" divide a node into layers.

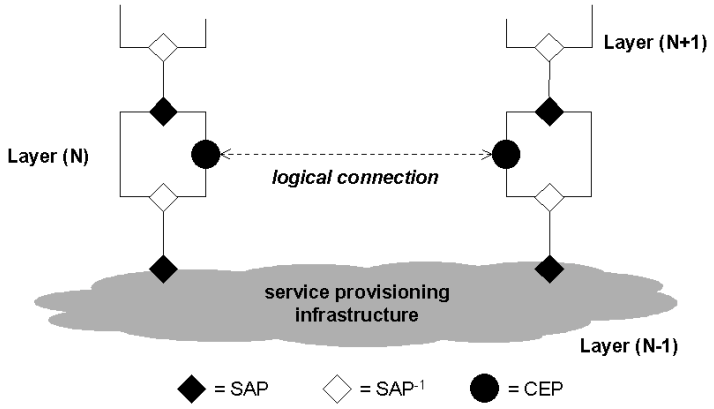


Figure 1: A simplified communication model based on OSI RM

Consequently, three main interfaces completely describe the behavior of a node layer from an outer perspective, each interface covering a specific aspect of the communication relation, see figure 1. The SAP, denoted by a filled diamond symbol, provides layer (N) services by means of so-called *service primitives* to a service user, the upper layer $(N+1)$. Service primitives can be implemented as procedure calls, library calls, signals, methods etc., which is a design decision. The CEP, symbolized by a filled circle, describes the "horizontal" relation to another remote peer entity. A CEP holds the specification of a communication protocol such as the Transmission Control Protocol (TCP) [19] or the Internet Protocol (IP) [18]. In fact, we will exemplify the topic of discussion on TCP. Be aware that the CEP is purely virtual and represents a logical interface only. All protocol messages are transmitted using the services of a lower layer. This interface function is given by the inverse SAP (SAP^{-1}), which uses services from a lower layer $(N-1)$ by accessing the $(N-1)$-SAP; it is depicted by an "empty" diamond symbol.

The model described bases on the OSI (Open Systems Interconnection) Reference Model [12], which has laid a solid foundation for understanding distributed system intercommunication [3]. The notation used for the SAP and SAP^{-1} is an extension to ROOM; for a thorough discussion see [8].

1.2 The Problem

Given the model presented, one faces some important problems in modeling the behavior of a layer in a communication system. There are in principle two alternatives for specifying layer (N). For this discussion, we assume that Finite State Machines (FSM) according to the Unified Modeling Language (UML) [17] are the primary means to describe behavioral aspects. FSMs are a common tool for specifying protocols [11].

- **Black box view:** Specifying a layer in a *black box* manner means that we give a complete description of the behavior of each and every external interface. In that case, the CEP, the SAP, and the SAP^{-1} are specified by an FSM each, which is a precise description of the remote peer protocol and the two interface protocols. Even though this view is ideal from a modeling point of view, the problem is that such a black box model can neither simulate nor explain the interface interaction without being wired with the internal behavior.

- **White box view:** Specifying a layer in a *white box* manner means that we define a more or less huge and complex FSM that gives a complete specification of the internals driving the external behavior. As a result, the communication at an external interface cannot be understood without looking inside the layer; at best, a list of messages (or service primitives) going in and out at the external interface can be declared. This corresponds to the notion of an interface in UML, see e.g. [4, p.155ff.]. Here, the problem is that the FSM is difficult to structure in a way, so that at least internally the behavioral aspects of the external interfaces are made clear.

What both problems have in common is that different views change scope and redefine how states or state machines are coupled with each other. In case of white box specifications, the UML offers the concept of composite states, which can be decomposed into two or more concurrent substates, also called *regions*. In order to enable synchronization and coordination of regions, the UML introduced *synch states*. However, synch states do not sufficiently support enough synchronization means as the case study presented below shows, nor do they solve the problem of synchronizing states of distinct state machines. Driven by a black box view, we propose the idea of Trigger Detection Points (TDP) to enable FSM separation but smooth coupling. TDPs together with Trigger Initiation Points (TIP) are introduced as a concept extending state machine modeling; they were motivated by the concept of detection points in [1].

2 Case Study: The TCP Communication Layer

The TCP protocol serves as an excellent example for discussing layer design and specification problems. It is simple to understand, easy to read (the technical standard [19] sums up to less than one hundred pages)[1], public available, and – most important – it is widespread and one of the most used protocols world-wide. Together with IP, the TCP/IP protocol suite forms the backbone of the Internet architecture.

Looking at how the TCP standard [19] specifies the protocol unveils a typical problem: It presents the whole layer by a state machine and does not clearly separate the TCP protocol from its user (or application) interface. Both are combined, see figure 2; it is the result of a white box view. The figure uses a compact notation and shows both the server FSM and the client FSM. It reads as follows: When a user in his role as a server submits a LISTEN command, the state changes from CLOSED to LISTEN. If, on the other side, the client user

[1] Clarifications and bug fixes are detailed in [5], extensions are given in [14].

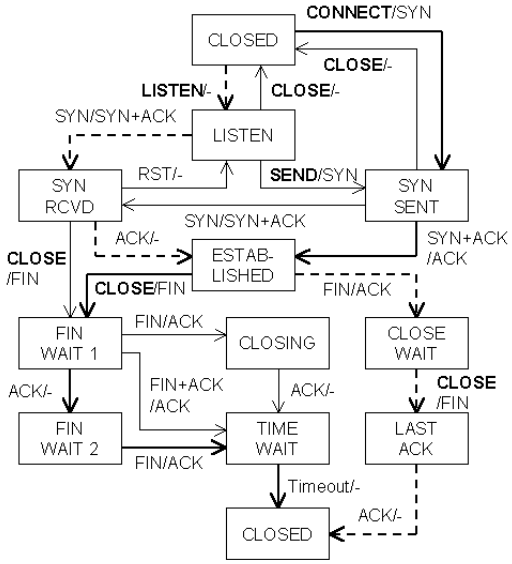


Figure 2: The TCP FSM figure is derived from [21, p.532]. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. User commands are given in bold font.

submits a CONNECT, the TCP protocol sends out a message with the synchronization bit SYN set to one, and the client's state changes to SYN SENT. On receipt of the TCP message with SYN equal to one, the server sends out a TCP message with SYN and ACK (the acknowledgment bit) set to one and changes to state SYN RCVD. When the three-way handshake completes successfully, both parties end up in state ESTABLISHED and are ready to send and receive data respectively. This short description of figure 2 neglects a lot of details of TCP (e.g. timeouts, which are important to resolve deadlocks and failures) but is sufficient for the purpose of our discussion. The interested reader may consult [21] for more information.

In order to structure TCP according to its interface functions (figure 1) the FSM in figure 2 needs to be partitioned. The result of this step is shown in figure 3 in UML notation. The left hand side of figure 3 displays the FSM, which corresponds in functionality to the SAP. Instead of using the TCP service commands LISTEN, CONNECT, SEND etc., the commands have been converted to service primitives, which are more narrative. Again, the client and the server side are combined in a single SAP FSM. From a user's viewpoint the communication with the client/server SAPs looks like follows: When a user requests a connection (Conn.req), the client's SAP changes to state C-PENDG. The server gets notified by the connection request via a connection indication (Conn.ind) and may respond with Conn.res, accepting the request. This is confirmed to the client via Conn.con and finally, the SAPs end up in state DATA. Note that neither the user of the client SAP nor the user of the server SAP see the underlying TCP protocol being used. They only see the

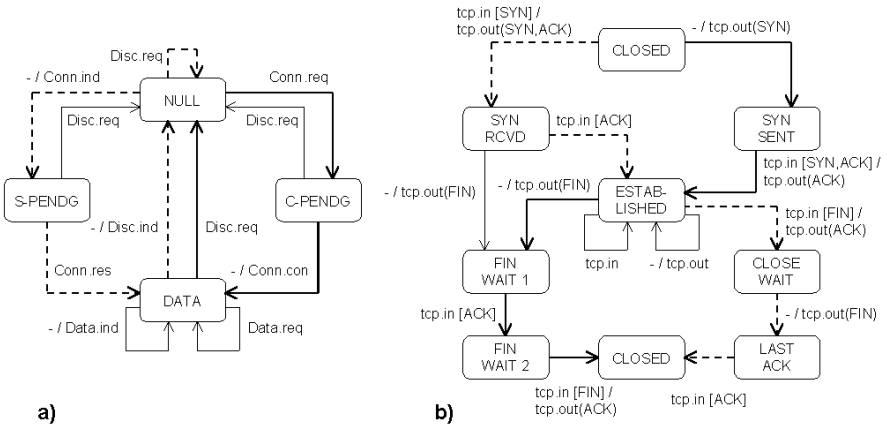


Figure 3: The FSMs of the SAP and the CEP of the TCP layer. The shortcuts stand for connect and disconnect; the postfixes stand for request, confirmation, indication, and response

SAP interface; the layer and its use of TCP is hidden.

The logical CEP holds the protocol specification of TCP, see the right hand side of figure 3. Since we have not introduced any coupling yet, the CEP FSM is strictly separated from the SAP FSM. That is why there is for example no indication what might have triggered the transition from CLOSED to SYN SENT at the client's side; but when the transition is triggered, no matter how it happened, then it sends out a TCP message with the SYN bit set. Otherwise, figure 3b is similar to figure 2; just all the numerous SAP related details have been stripped off. To reduce complexity, we slightly simplified the TCP protocol specification and added transitions to the data transfer state ESTABLISHED.

As described, TCP calls on a lower level protocol module to actually send and receive information over a network. This lower level protocol module usually is IP and is accessed via the inverse TCP SAP^{-1}. To avoid cluttering up the discussion and distracting the reader with too many FSMs, we intentionally left out an example figure. For the sake of brevity, the SAP^{-1} is not considered and supposed not to exist; that is we assume the logical connection between the CEPs to be for real. Consequently, we can restrict the discussion on the interaction between the SAP and the CEP; this simplifies and eases the topic under discussion.

3 The Concept of Coupled State Machines

We managed to partition TCP according to its layer interfaces, which already is an achievement. All further details of TCP like flow control and buffering, congestion control, fragmentation, error control, window flow control etc. are hidden and subject of a refined view. As was mentioned above: If we prefer a white box view, the two state machines could be

interpreted as concurrent regions in a “higher level” statechart and synchronized via synch states. If we, on the opposite, demand a rigid black box view (as is often the case for architectural modeling), the SAP and the CEP are described by two separate FSMs specifying the “horizontal” and “vertical” communication behavior; there are no coupling capabilities. However, for model understanding it would be beneficial to show how the different interfaces of the communication layer interact with each other without referring to any internals. As was shown by Ericsson’s language study, it is usually the “inside”, which drives the “outside”. We are looking for a way that allows the modeler to keep a purely external view.

One way to couple the individual FSMs is by the usual event messaging mechanism provided by UML, that means by signals and/or call events. The drawback of this approach is that one would again tightly connect the FSMs. For example, the Conn.req transition of the SAP (see figure 3a) needs to have an activity attached that sends a signal to the CEP (see figure 3b). This signal would then represent the CLOSED/SYN SENT transition that triggers the tcp.out message. As a result, the FSM of the CEP would more or less turn out to be the original TCP FSM and finally look like figure 2. In other words, the modeler would not be better off, and splitting of the TCP FSMs seems to be an academic exercise only. Obviously, another technique is needed.

Our solution to this problem is the introduction of so-called Trigger Detection Points (TDPs) and Trigger Initiation Points (TIPs). A TDP can be attached at the arrow head of an transition in a statechart diagram; it detects whenever this specific transition fires and broadcasts a notification message to all corresponding TIPs. TDPs are notated by small filled boxes, see figure 4. A TIP can be attached at the beginning of the transition arrow and triggers the transition to fire. An *active* TIP stimulates the transition to fire on receipt of a TDP notifier independent of the transition’s event-signature. That means, that either the event specified by the transition’s event-signature *or* the TIP can trigger the transition. Active TIPs are visualized by small filled triangles, see figure 4. *Passive* TIPs, on the other hand, have a locking mechanism and can be meaningfully used with “normal” transitions only, i.e. the transition explicitly requires an event-signature. The transition cannot fire unless the TIP’s corresponding TDP has been passed *and* unless the transition’s event has been received. The order of occurrence is irrelevant, it is just the combination of the TIP event *and* the transition event, which unlock the transition and let it fire. Passive TIPs behave like a logical “and” to synchronize a transition, whereas active TIPs realize a logical “or”. An example of a passive TIP can be found in figure 4a; it is pictured by a small, “empty” triangle. In general, the relation of a TIP and a TDP is given by a name consisting of a single or more capital letters. Note that one or more TIPs may be related to a single TDP.

Now, the coupling of the SAP and the CEP can be easily described, see figure 4a and 4b. For example, when a client user sends a Conn.req to the SAP, TDP A detects the transition NULL to C-PENDG firing and broadcasts a notifier event to all corresponding TIPs. The notifier event causes the CEP to fire the CLOSED/SYN SENT transition and results in sending out a TCP message with the SYN bit set to one; the rest of the scenario is straight forward. However, some explanations should help understand the purpose of a passive TIP. Let us assume, that the protocol at the server side has just entered state SYN RCVD,

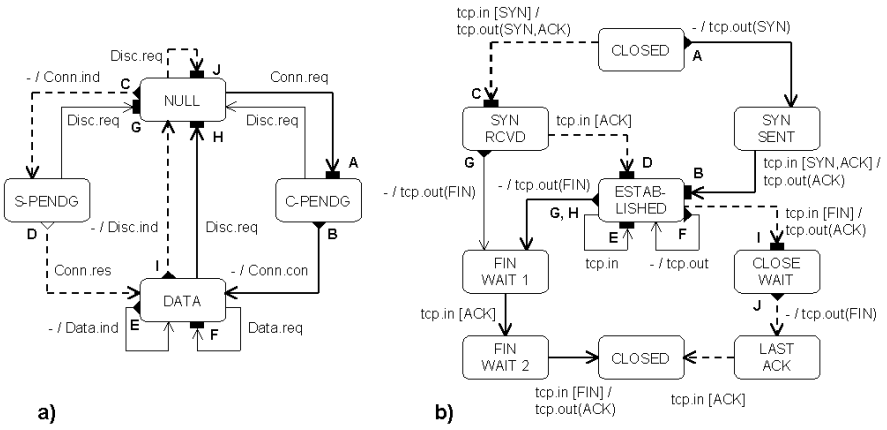


Figure 4: The FSMs of the SAP and the CEP of the TCP layer coupled via TDPs and TIPs

which triggers TIP C at the server SAP and results in a connection indication (Conn.ind) to the SAP user. Now, there are two concurrent and competing threads. The user of the server SAP may either accept the connection indication and answer with Conn.res or, alternatively, the user may deny the request and answer with a Disc.req. Concurrently, on the protocol thread, the server's CEP enters state ESTABLISHED at some point in time. It is the passive TIP D that prevents the SAP FSM entering DATA on Conn.req unless the protocol has reached ESTABLISHED. On the other hand, if the user has decided to reject the connection indication (Conn.ind) via Disc.req, the CEP starts the disconnect procedure based on the TDP G trigger. All this could not be done using conventional messaging without changing the FSMs.

The advantage of using TDPs and TIPs is that the FSMs remain autonomous but get coupled. They can notify each other about important state changes and use it for synchronization purposes; there is no need to introduce new event messages and modify transitions. TDPs and TIPs could be interpreted as some sort of annotations (with precise semantics), which specify FSM interaction and coordination. The modeler does not need to modify the original interface specification or reference to any internal "engine" driving the whole. If the broadcasting mechanism of TDP events can be directed, it is possible to couple external interface FSMs with layer internal FSMs reusing the same set of TDPs and TIPs. That means, that a black box and a white box view could peacefully coexist without blurring the difference between both views.

4 Extending the UML

Synch states as known from the UML correspond in their behavior to what we called *passive* TIPs: A synch state is used in conjunction with forks and joins to insure that one

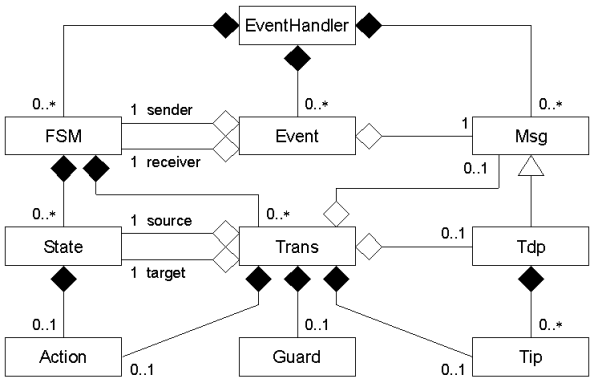


Figure 5: The design of the FSM prototype

region leaves a particular state or states before another region can enter a particular state or states [17]. Clearly, synch states do not support other synchronization means between regions like TIPs do and they are not suited for inter-FSM synchronization. Good reasons to think about integrating TIPs and TDPs in the UML and to substitute synch states.

TDPs and TIPs can be smoothly integrated in an event driven execution model for FSMs. The prototype we developed at Ericsson (programmed in Python [16]) treats TDPs as a specialization of messages, see figure 5, and dispatches notifier events to the event queue. The implementation of TIPs required only a few modifications to the event processor.

If one compares the prototype design and the metamodel for state machines (see section 2.12 of the UML [17] semantics), the required extensions to the UML can be easily identified: First, the notifier event needs to be subclassed to the event metaclass;[2] this can be achieved by using stereotypes. Then, it is to be decided how TDPs can be attached to the transition metaclass. Since transitions are restricted to have not more than one event trigger, it is not possible to add TDPs as a second trigger. Rather, the transition metaclass can be extended by some few properties. A TDP property is needed referring to the notifier event, optionally added by a property holding a list of state machines the notifier event is selectively broadcasted to. Another property are the TIP and the TIP type, which hold the notifier reference and the value active or passive, respectively. The required changes to the execution semantics of state machines are uncritical, since the UML is relatively open to adaptations. To conclude, the extensions described are the simplest form to introduce TDPs and TIPs to the UML using its extension mechanisms [10].

Note that TDPs and TIPs make synch states superfluous. TDPs/TIPs contain the concept of synch states but allow much more semantic variations and extensions. Synch states are an oddity in the UML with no clear conceptual roots; TDPs and TIPs are their generalization but they are put in a meaningful semantical context of transitions and events. In fact, TIPs and TDPs specify a synchronization protocol between states machines or regions. Such a protocol does not only seem more appropriate to capture complex interactions of

[2]Regarding events, the UML is a bit different designed than our message based prototype.

synchronization but also semantically cleaner. That is why we propose to remove synch states from the UML metamodel and instead introduce the notifier event subclass, insert metaclasses for TDPs and TIPs and associate them to the transition metaclass. This would enable flexible semantic extensions via stereotypes to the UML user.

5 Conclusions

Actually, the TDP/TIP concept relates very much to the observer pattern [6]; it allows the modeler to notify other FSMs about state changes. Because of the distinction in active and passive TIPs, the concept of coupled state machines implements an extended observer pattern. This lifts the observer pattern from its use in the design domain in form of class diagrams to the modeling domain with an explicit notation for coupling, which is a quite interesting aspect. Furthermore, it is an interesting question, if TIPs and TDPs could be of use in sequence diagrams or Message Sequence Charts (MSC) [13].

Since the approach presented gives means to specify and separate aspects of a modeling entity, one could also investigate to which extend TDPs and TIPs enable aspect-oriented modeling in extension to aspect-oriented programming [15]. It also allows the modeler to specify APIs (Application Programming Interface) much more elegant; for instance, the TCP SAP could be seen as an API to TCP. As was shown in the case study, the design of communication protocols gains a lot of clarity from the separation of logical concerns. In short, it looks like that many application areas could benefit from using coupled state machines.

Due to the specific nature of the application domain (data and telecommunications) we study, we cannot claim that we have identified all types of TDPs and TIPs required for coupling FSMs in an efficient manner. Extensions or specializations are conceivable. However, TDPs and TIPs appear to be a powerful modeling concept, they substitute synch states, and put a modeler in a better position especially for modeling the coordination and synchronization of concurrent systems.

Acknowledgements: Many thanks to Andreas Witzel, who triggered the idea of coupled state machines. Furthermore, the authors would like to thank Jörg Bruß and Dietmar Wenninger (all Ericsson) for their support.

References

[1] Customised Applications for Mobile network Enhanced Logic (CAMEL) Phase 3 – Stage 2. Technical Specification 3G TS 23.078 version 3.1.0, 3rd Generation Partnership Project, Valbonne, France, August 1999.

[2] Helmut Balzert. *Lehrbuch der Software-Technik: Software Entwicklung*. Spektrum Akademischer Verlag, 1996.

[3] Hans Wilhelm Barz. *Kommunikation und Computernetze: Konzepte, Protokolle und Standards*. Hanser, 1991.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[5] Robert Braden. Requirements for Internet Hosts – Communication Layers. Standard RFC 1122, Internet Engineering Task Force, October 1989.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Dominikus Herzberg. UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain. In Robert France and Bernhard Rumpe, editors, *≪UML≫'99 – The Unified Modeling Language: Beyond the Standard; Second International Conference, Fort Collins, CO, USA, October 28–30, 1999*, LNCS 1723, pages 330–338. Springer, 1999.

[8] Dominikus Herzberg and André Marburger. The Use of Layers and Planes for Architectural Design of Communication Systems. In *The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg, Germany; May 2–4, 2001*, pages 235–242. IEEE Computer Society, May 2001.

[9] Dominikus Herzberg, André Marburger, and Tony Jokikyyny. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. 2. Workshop Software-Reengineering, Bad Honnef, Germany, 11.-12. May, 2000.

[10] Dominikus Herzberg and Lars von Wedel. Erweiterungsmechanismen der UML. *OBJEKTspektrum*, (4):56–59, Juli/August 1999.

[11] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[12] Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ITU-T Recommendation X.200, International Telecommunication Union, July 1994.

[13] Message Sequence Chart (MSC). ITU-T Recommendation Z.120, International Telecommunication Union, November 1999.

[14] Van Jacobson, Bob Braden, and Dave Borman. TCP Extensions for High Performance. Standard RFC 1323, Internet Engineering Task Force, May 1992.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer, June 1997.

[16] Mark Lutz. *Programming Python*. O'Reilly, 1996.

[17] Unified Modeling Language Specification, Version 1.4. Technical Specification, Object Management Group (OMG), February 2001.

[18] Internet Protocol. Standard RFC 791, Internet Engineering Task Force, September 1981.

[19] Transmission Control Protocol. Standard RFC 793, Internet Engineering Task Force, September 1981.

[20] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[21] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 3rd edition, 1996.